



Python Interface

Release 4.0

Luis Saavedra

March 22, 2010

CONTENTS

1	Introduction	1
2	Installation	3
2.1	In a debian/ubuntu system	3
2.2	In a general unix/linux based systems	3
3	Preliminary	7
4	<i>Python GetFEM++</i> interface	9
4.1	Introduction	9
4.2	Memory Management	9
4.3	Documentation	9
4.4	<i>Python GetFEM++</i> organization	10
5	Examples	13
5.1	A step-by-step basic example	13
5.2	Another Laplacian with exact solution (source term)	16
5.3	Final Laplacian with exact solution (Neumann boundary condition)	17
5.4	Linear and non-linear elasticity	19
5.5	Avoiding the bricks framework	22
5.6	Other examples	23
6	How-tos	25
6.1	Import gmsh mesh	25
7	API reference	27
7.1	CvStruct	27
7.2	Eltn	27
7.3	Fem	28
7.4	GeoTrans	30
7.5	GlobalFunction	31
7.6	Integ	31
7.7	LevelSet	33
7.8	MdBrick	34
7.9	MdState	38
7.10	Mesh	39
7.11	MeshFem	44
7.12	MeshIm	48
7.13	MeshLevelSet	50
7.14	Model	50

7.15	Precond	57
7.16	Slice	58
7.17	Spmat	62
7.18	Module asm	65
7.19	Module compute	67
7.20	Module delete	69
7.21	Module linsolve	69
7.22	Module poly	69
7.23	Module undelete	70
7.24	Module util	70
Index		71

INTRODUCTION

This guide provides a reference about the *Python* interface of *GetFEM++*. For a complete reference of *GetFEM++*, please report to the [specific guides](#), but you should be able to use the python interface without any particular knowledge of the *GetFEM++* internals, although a basic knowledge about Finite Elements is required.

Copyright © 2000-2010 Yves Renard, Julien Pommier.

The text of the *GetFEM++* website and the documentations are available for modification and reuse under the terms of the [GNU Free Documentation License](#)

The program *GetFEM++* is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; version 2.1 of the License. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

INSTALLATION

2.1 In a debian/ubuntu system

If you have a problem installing the packages, please report it as a bug.

Edit `/etc/apt/sources.list` and add the following lines:

```
deb http://apt.dim.uchile.cl distro main
deb-src http://apt.dim.uchile.cl distro main
```

where

`distro = debian xor ubuntu,`

then, do a `aptitude update` and `aptitude install python-getfem`.

2.2 In a general unix/linux based systems

Since we use standard *GNU* tools, the installation of the *Python GetFEM++* is somewhat standard.

2.2.1 Requirements

It requires the python developpement files (`python.h` etc.) to be available (package *python-all-dev* in debian distribution), and also the numpy and scipy packages to be installed (package *python-numpy* and *python-scipy* in debian distribution). In case of troubles with a non-GNU compiler, `gcc/g++ (>= 4.1)` should be a safe solution (package *build-essential* in debian distribution).

If you want mesh generation, it requires the package `qhull` installed on your system (package *libqhull-dev* in debian distribution).

If you want to build binaries from svn to get the latest changes, improvements, bugfixes, new bugs, etc. It requires an svn client, automake, and libtool.

2.2.2 Download sources

There are two ways to get *GetFEM++*, either as a compressed package (stable release) or via anonymous svn access (unstable releases).

The latest stable release of *GetFEM++* is `getfem++-1version1tar.gz`

Inline substitution_reference start-string without end-string.

- download package:

```
wget http://download.gna.org/getfem/stable/getfem++-!version!tar.gz
```

Inline substitution_reference start-string without end-string.

- unpack:

```
tar xzf getfem++-!version!tar.gz
```

Inline substitution_reference start-string without end-string.

- and go to the root directory of getfem:

```
cd getfem++-4.0.0.
```

The latest unstable releases is:

- checkout over SVN protocol (TCP 3690):

```
svn co svn://svn.gna.org/svn/getfem/trunk getfem
```

- or checkout over HTTP protocol (TCP 80):

```
svn co http://svn.gna.org/svn/getfem/trunk getfem
```

- go to the root directory of getfem:

```
cd getfem/getfem++
```

- and run `autogen.sh` script:

```
bash autogen.sh
```

2.2.3 Compiling

Configure with:

```
./configure --enable-python=yes
```

If you want to use a specific **BLAS** library, you may have to supply the necessary link flags and libs to the configure script, for example with:

```
./configure --enable-python=yes BLAS_LIBS="-L/usr/lib/sse2/atlas -lblas"
```

More specific instructions can be found in the README* files of the distribution.

Warning:

- you should not use a different compiler than the one that was used for *GetFEM++*.
- you should have built the *GetFEM++* static library (i.e. do not use `./configure --disable-static` when building *GetFEM++*).
- On linux/x86_64 platforms, a mandatory option when building *GetFEM++* (and any static library linked to them) is the `-with-pic` option of their `./configure` script.

Then start the compilation with:

make

and finally install with:

make install

PRELIMINARY

This is just a short summary of the terms employed in this manual. If you are not familiar with finite elements, this should be useful (but in any case, you should definitively read the *Introduction to the finite element kernel* (in the project documentation) and the *getfem++ user manual* in the [specific guides](#).

The **mesh** is composed of **convexes**. What we call convexes can be simple line segments, prisms, tetrahedrons, curved triangles, or even something which is not convex (in the geometrical sense). They all have an associated **reference convex**: for segments, this will be the $[0, 1]$ segment, for triangles this will be the canonical triangle $(0, 0) - (0, 1) - (1, 0)$, etc. All convexes of the mesh are constructed from the reference convex through a **geometric transformation**. In simple cases (when the convexes are simplices for example), this transformation will be linear (hence it is easily inverted, which can be a great advantage). In order to define the geometric transformation, one defines **geometrical nodes** on the reference convex. The geometrical transformation maps these nodes to the **mesh nodes**.

On the mesh, one defines a set of basis functions: the **FEM**. A FEM is associated at each convex. The basis functions are also attached to some geometrical points (which can be arbitrarily chosen). These points are similar to the mesh nodes, but *they don't have to be the same* (this only happens on very simple cases, such as a classical P1 fem on a triangular mesh). The set of all basis functions on the mesh forms the basis of a vector space, on which the PDE will be solved. These basis functions (and their associated geometrical point) are the *degrees of freedom* (contracted to **dof**). The FEM is said to be **Lagrangian** when each of its basis functions is equal to one at its attached geometrical point, and is null at the geometrical points of others basis functions. This is an important property as it is very easy to **interpolate** an arbitrary function on the finite elements space.

The finite elements method involves evaluation of integrals of these basis functions (or product of basis functions etc.) on convexes (and faces of convexes). In simple cases (polynomial basis functions and linear geometrical transformation), one can evaluate analytically these integrals. In other cases, one has to approximate it using **quadrature formulas**. Hence, at each convex is attached an **integration method** along with the FEM. If you have to use an approximate integration method, always choose carefully its order (i.e. highest degree of the polynomials who are exactly integrated with the method): the degree of the FEM, of the polynomial degree of the geometrical transformation, and the nature of the elementary matrix have to be taken into account. If you are unsure about the appropriate degree, always prefer a high order integration method (which will slow down the assembly) to a low order one which will produce a useless linear-system.

The process of construction of a global linear system from integrals of basis functions on each convex is the **assembly**.

A mesh, with a set of FEM attached to its convexes is called a **mesh_fem** object in *GetFEM++*.

A mesh, with a set of integration methods attached to its convexes is called a **mesh_im** object in *GetFEM++*.

A *mesh_fem* can be used to approximate scalar fields (heat, pression, ...), or vector fields (displacement, electric field, ...). A *mesh_im* will be used to perform numerical integrations on these fields. Most of the finite elements implemented in *GetFEM++* are scalar (however, TR0 and edges elements are also available). Of course, these scalar FEMs can be used to approximate each component of a vector field. This is done by setting the **Qdim** of the *mesh_fem* to the dimension of the vector field (i.e. $Qdim=1 \Rightarrow$ scalar field, $Qdim=2 \Rightarrow$ 2D vector field etc...).

When solving a PDE, one often has to use more than one FEM. The most important one will be of course the one on

which is defined the solution of the PDE. But most PDEs involve various coefficients, for example:

$$\nabla \cdot (\lambda(x) \nabla u) = f(x).$$

Hence one has to define a FEM for the main unknown u , but also for the data $\lambda(x)$ and $f(x)$ if they are not constant. In order to interpolate easily these coefficients in their finite element space, one often choose a Lagrangian FEM.

The convexes, mesh nodes, and dof are all numbered. We sometimes refer to the number associated to a convex as its **convex id** (contracted to **cvid**). Mesh node numbers are also called **point id** (contracted to **pid**). Faces of convexes do not have a global numbering, but only a local number in each convex. Hence functions which need or return a list of faces will always use a two-rows matrix, the first one containing convex ids, and the second one containing local face number.

While the dof are always numbered consecutively, *this is not always the case for point ids and convex ids*, especially if you have removed points or convexes from the mesh. To ensure that they form a continuous sequence (starting from 1), you have to call `m.set('optimize structure')`.

PYTHON GETFEM++ INTERFACE

4.1 Introduction

GetFEM++ provides an interface to the *Python* scripting language. *Python* is a nice, cross-platform, and free language. With the addition of the *numpy* package, python provides a subset of Matlab functionalities (i.e. dense arrays). The *VTK* toolkit may provide visualization tools via its python interface (or via *MayaVi*), and data files for *OpenDX* may be exported. In this guide, nevertheless, to visualize the results, we will export to *Gmsh* post-processing format. The sparse matrix routines are provided by the *getfem* interface.

The python interface is available via a python module *getfem.py*. In order to use the interface you have to load it with:

```
import getfem
m = getfem.Mesh('cartesian', range(0, 3), range(0,3))
```

or:

```
from getfem import *
m = Mesh('cartesian', range(0, 3), range(0,3))
```

If the *getfem.py* (and the internal *_getfem.so*) module is not installed in a standard location for python, you may have to set the *PYTHONPATH* environment variable to its location. For example with:

```
import sys
sys.path.append('.../getfem/getfem++/interface/src/python/')
```

4.2 Memory Management

A nice advantage over the Matlab interface is that you do not have to explicitly delete objects that are not used any more, this is done automatically. You can however inspect the content of the *getfem* workspace with the function `getfem.memstats()`.

4.3 Documentation

The *getfem* module is largely documented. This documentation has been extracted into the *API reference*. The *getfem-matlab* user guide may also be used, as 95% of its content translates quite directly into python (with the exception of the plotting functions, which are specific to matlab).

4.4 Python GetFEM++ organization

The general organization of the python-interface is the following:

- Each class from the matlab interface has a corresponding class in the python interface: the gfMesh class becomes the getfem.Mesh class in python, the gfSlice becomes the getfem.Slice etc.
- Each get and set method of the matlab interface has been translated into a method of the corresponding class in the python interface. For example:

```
gf_mesh_get(m, 'outer faces');  
gf_mesh_get(m, 'pts');
```

becomes:

```
m.outer_faces();  
m.pts();
```

Some methods have been renamed when there was ambiguity, for example `gf_mesh_set(m, 'pts', P)` is `m.set_pts(P)`.

- The other getfem-matlab function have a very simple mapping to their python equivalent:

<code>gf_compute(mf,U,'foo',...)</code>	<code>getfem.compute_foo(mf,U)</code> or <code>getfem.compute('foo',...)</code>
<code>gf_asm('foobar',...)</code>	<code>getfem.asm_foobar(...)</code> or <code>getfem.asm('foobar',...)</code>
<code>gf_linsolve('gmres',...)</code>	<code>getfem.linsolve_gmres(...)</code> or <code>getfem.linsolve('gmres',...)</code>

class CvStruct (*self*, *args)

Descriptor for a convex structure objects, stores formal information convex structures (nb. of points, nb. of faces which are themselves convex structures)

class GeoTrans (*self*, *args)

Descriptor for geometric transformations objects (defines the shape/position of the convexes).

class Mesh (*self*, *args)

Descriptor for mesh structure (nodes, convexes, geometric transformations for each convex).

class Fem (*self*, fem_name)

Descriptor for FEM (Finite Element Method) objects (one per convex, can be PK, QK, HERMITE, etc...).

class Integ (*self*, *args)

Descriptor for Integration Method objects (exact, quadrature formulaldots). Although not linked directly to GeoTrans, an integration method is usually specific to a given convex structure.

class MeshFem (*self*, *args)

Descriptor for object linked to a mesh, where each convex has been assigned a FEM.

class MeshIm (*self*, *args)

Descriptor for object linked to a mesh, where each convex has been assigned an integration method.

class Model (*self*, *args)

Descriptor for *model* object, holds the global data, variables and description of a model. Evolution of *model state* and *model brick* object for 4.0 version of GetFEM++.

class MdState (*self*, *args)

Descriptor for *model state* object, holds the global data for a stack of MdBrick (global tangent matrix, right hand side etc.). **Deprecated** since 4.0 version of GetFEM++, see *model* object.

class MdBrick (*self*, *args)

Descriptor for *model brick* object, an abstraction of a part of solver (for example, the part which build the tangent

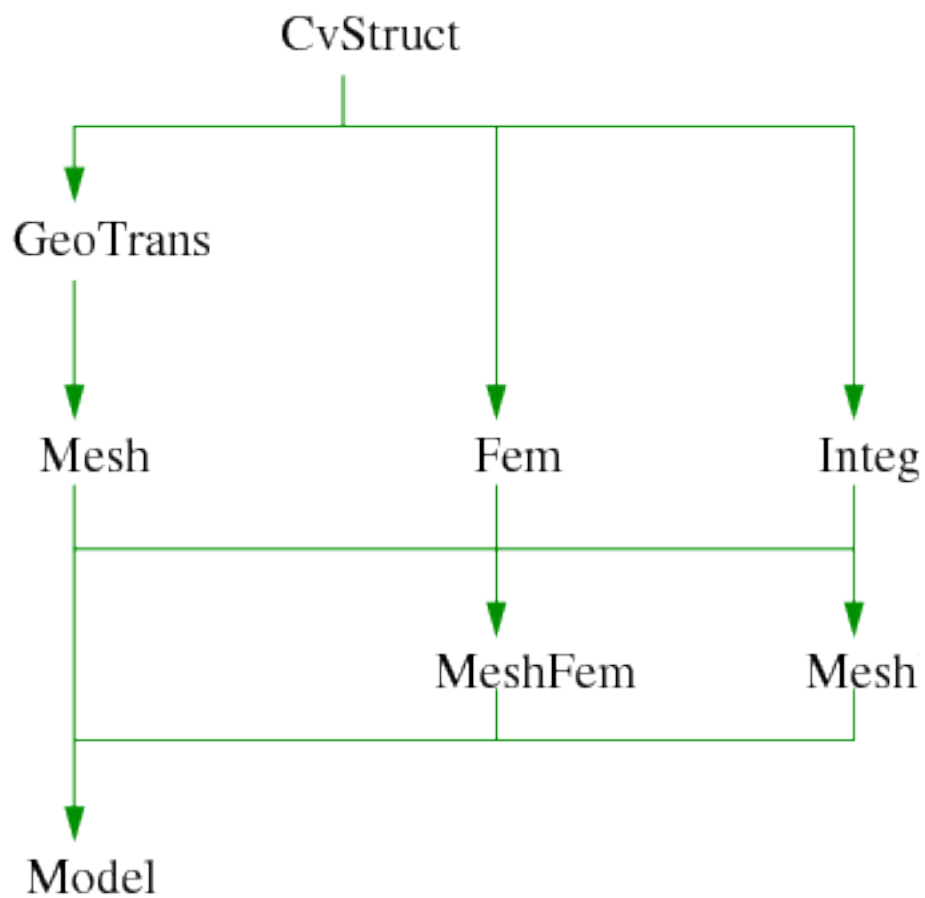


Figure 4.1: python-getfem interface objects hierarchy.

matrix, the part which handles the dirichlet conditions, etc.). These objects are stacked to build a complete solver for a wide variety of problems (they typically use a number of MeshFem, MeshIm etc.). **Deprecated** since 4.0 version of *GetFEM++*, see *model* object.

EXAMPLES

5.1 A step-by-step basic example

This example shows the basic usage of `getfem`, on the über-canonical problem above all others: solving the Laplacian, $-\Delta u = 0$ on a square, with the Dirichlet condition $u = g(x)$ on the domain boundary.

The first step is to *create a Mesh object*. Since *GetFEM++* does not come with its own mesher, one has to rely on an external mesher (see `getfem.Mesh('import', string FORMAT, string FILENAME)`), or use very simple meshes. For this example, we just consider a regular mesh whose nodes are $\{x_{i=0\dots 10, j=0\dots 10} = (i/10, j/10)\}$

```
# import basic modules
import getfem as gf
import numpy as np

# creation of a simple cartesian mesh
m = gf.Mesh('cartesian', np.arange(0,1.1,0.1), np.arange(0,1.1,0.1))
```

The next step is to *create a MeshFem object*. This one links a mesh with a set of FEM

```
# create a MeshFem of for a field of dimension 1 (i.e. a scalar field)
mf = gf.MeshFem(m, 1)
# assign the Q2 fem to all convexes of the MeshFem
mf.set_fem(gf.Fem('FEM_QK(2,2)'))
```

The first instruction builds a new `MeshFem` object, the second argument specifies that this object will be used to interpolate scalar fields (since the unknown u is a scalar field). The second instruction assigns the Q^2 FEM to every convex (each basis function is a polynomial of degree 4, remember that $P^k \Rightarrow$ polynomials of degree k , while $Q^k \Rightarrow$ polynomials of degree $2k$). As Q^2 is a polynomial FEM, you can view the expression of its basis functions on the reference convex

```
# view the expression of its basis functions on the reference convex
print gf.Fem('FEM_QK(2,2)').poly_str()
```

Now, in order to perform numerical integrations on `MeshFem`, we need to *build a MeshIm object*

```
# an exact integration will be used
mim = gf.MeshIm(m, gf.Integ('IM_EXACT_PARALLELEPIPED(2)'))
```

The integration method will be used to compute the various integrals on each element: here we choose to perform exact computations (no quadrature formula), which is possible since the geometric transformation of these convexes from the reference convex is linear (this is true for all simplices, and this is also true for the parallelepipeds of our

regular mesh, but it is not true for general quadrangles), and the chosen FEM is polynomial. Hence it is possible to analytically integrate every basis function/product of basis functions/gradients/etc. There are many alternative FEM methods and integration methods (see [the description of finite element and integration methods](#)).

Note however that in the general case, approximate integration methods are a better choice than exact integration methods.

Now we have to *find the <boundary> of the domain*, in order to set a Dirichlet condition. A mesh object has the ability to store some sets of convexes and convex faces. These sets (called <regions>) are accessed via an integer *id*

```
# detect the border of the mesh
border = m.outer_faces()
# mark it as boundary #42
m.set_region(42, border)
```

Here we find the faces of the convexes which are on the boundary of the mesh (i.e. the faces which are not shared by two convexes). The array `border` has two rows, on the first row is a *convex ids*, on the second row is a *face ids* (which is local to the convex, there is no global numbering of faces). Then this set of faces is assigned to the region number 42.

At this point, we just have to stack some model bricks and run the solver to get the solution! The *model bricks* are created with the `MdBrick` constructor. A model brick is basically an object which modifies a global linear system (tangent matrix for non-linear problems) and its associated right hand side. Typical modifications are insertion of the stiffness matrix for the problem considered (linear elasticity, laplacian, etc), handling of a set of constraints, Dirichlet condition, addition of a source term to the right hand side etc. The global tangent matrix and its right hand side are stored in a “model state” structure, created with the `MdState` constructor.

Let us build a problem with an easy solution: $u = x(x - 1) - y(y - 1)$, then we have $-\Delta u = 0$ (the FEM won't be able to catch the exact solution since we use a Q^2 method).

We start with a *generic elliptic* brick, which handles $-\nabla \cdot (A : \nabla u) = \dots$ problems, where A can be a scalar field, a matrix field, or an order 4 tensor field. By default, $A = 1$

```
# generic elliptic brick
b0 = gf.MdBrick('generic elliptic', mim, mf)
```

Each brick embeds a number of parameter fields. In the case of the generic elliptic brick, there is only one parameter field, the $A(x)$ coefficient in $-\nabla \cdot (A : \nabla u) = \dots$. It is possible to view the list of parameters of the brick with

```
# list of parameters of the brick
print b0.param_list()
```

Next we add a Dirichlet condition on the domain boundary

```
# add a Dirichlet condition on the domain boundary
b1 = gf.MdBrick('dirichlet', b0, 42, mf, 'penalized')
```

Here the number 42 is the region number to which the dirichlet condition is applied. The 'penalized' says that the Dirichlet condition should be imposed via a penalization technique. Other ways are possible ('augmented' and 'eliminated' systems). A `MeshFem` argument is also required, as the Dirichlet condition $u = r$ is imposed in a weak form $\int_{\Omega} u(x)v(x) = \int_{\Omega} r(x)v(x) \forall v$, where v is taken in the space of multipliers given here by `MeshFem`.

By default, the Dirichlet brick imposes $u = 0$ on the specified boundary. We change this to $u = x(x - 1) - y(y - 1)$

```
# change Dirichlet condition
R = mf.eval('x[0]*(x[0]-1) - x[1]*(x[1]-1)')
b1.set_param('R', mf, R)
```

Note: the polynomial expression was interpolated on MeshFem. It is possible only if MeshFem is of Lagrange type. In this first example we use the same MeshFem for the unknown and for the data such as R , but in the general case, `mf` won't be Lagrangian and another (Lagrangian) MeshFem will be used for the description of Dirichlet conditions, source terms etc.

A *model state* variable is created, and the solver is launched

```
# created model state
mds = gf.MdState('real')
bl.solve(mds)
```

The model state now contains the solution (as well as other things, such as the linear system which was solved). It is extracted

```
# extracted solution
sol = mds.get('state')
```

Then export solution

```
# export computed solution
mf.export_to_pos('sol.pos', sol, 'Computed solution')
```

and view with `gms sol.pos`

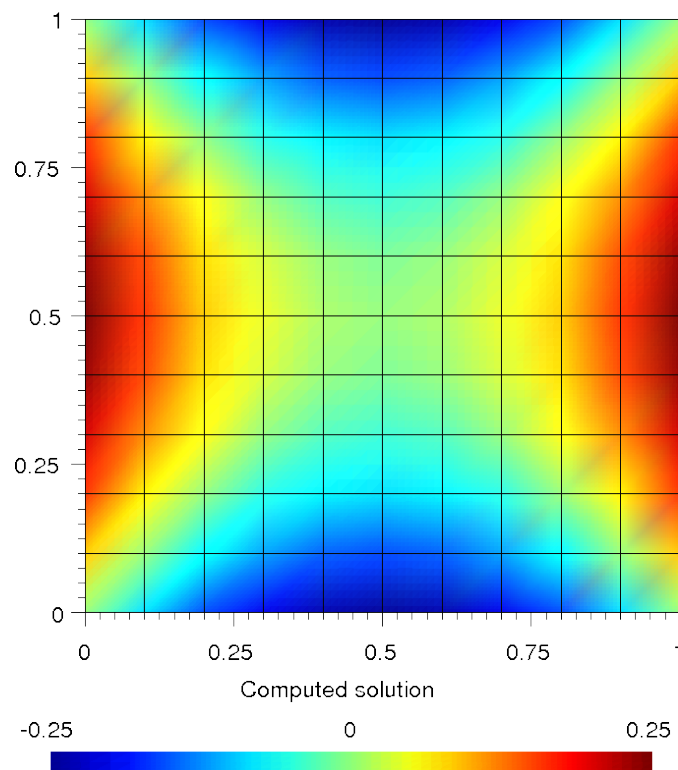


Figure 5.1: Computed solution

5.2 Another Laplacian with exact solution (source term)

This example shows the basic usage of getfem, on the canonical problem: solving the Laplacian, $-\Delta u = f$ on a square, with the Dirichlet condition $u = g(x)$ on the domain boundary Γ_D and the Neumann condition $\frac{\partial u}{\partial \eta} = h(x)$ on the domain boundary Γ_N .

We create Mesh, MeshFem, MeshIm object and find the boundary of the domain in the same way as the previous example

```
# import basic modules
import getfem as gf
import numpy as np

# boundary names
top    = 101 # Dirichlet boundary
down   = 102 # Neumann boundary
left   = 103 # Dirichlet boundary
right  = 104 # Neumann boundary

# parameters
NX = 40                                     # Mesh parameter
Dirichlet_with_multipliers = True;          # Dirichlet condition with multipliers or penalization
dirichlet_coefficient = 1e10;               # Penalization coefficient

# mesh creation
m = gf.Mesh('regular_simplices', np.arange(0,1+1./NX,1./NX), np.arange(0,1+1./NX,1./NX))

# create a MeshFem for u and rhs fields of dimension 1 (i.e. a scalar field)
mfu   = gf.MeshFem(m, 1)
mfrhs = gf.MeshFem(m, 1)
# assign the P2 fem to all convexes of the both MeshFem
mfu.set_fem(gf.Fem('FEM_PK(2,2)'))
mfrhs.set_fem(gf.Fem('FEM_PK(2,2)'))

# an exact integration will be used
mim = gf.MeshIm(m, gf.Integ('IM_TRIANGLE(4)'))

# boundary selection
flst   = m.outer_faces()
fnor   = m.normal_of_faces(flst)
ttop    = abs(fnor[1,:]-1) < 1e-14
tdown   = abs(fnor[1,:]+1) < 1e-14
tleft   = abs(fnor[0,:]+1) < 1e-14
tright  = abs(fnor[0,:]-1) < 1e-14
ftop    = np.compress(ttop, flst, axis=1)
fdown   = np.compress(tdown, flst, axis=1)
fleft   = np.compress(tleft, flst, axis=1)
fright  = np.compress(tright, flst, axis=1)

# mark it as boundary
m.set_region(top, ftop)
m.set_region(down, fdown)
m.set_region(left, fleft)
m.set_region(right, fright)
```

then, we interpolate the exact solution and source terms

```
# interpolate the exact solution (assuming mfu is a Lagrange fem)
G = mfu.eval('x[1]*(x[1]-1)*x[0]*(x[0]-1)+x[0]*x[0]*x[0]*x[0]*x[0]')

# interpolate the source terms (assuming mfrhs is a Lagrange fem)
F = mfrhs.eval('-(2*(x[0]*x[0]+x[1]*x[1])-2*x[0]-2*x[1]+20*x[0]*x[0]*x[0])')
H = mfrhs.eval('[x[1]*(x[1]-1)*(2*x[0]-1) + 5*x[0]*x[0]*x[0]*x[0], x[0]*(x[0]-1)*(2*x[1]-1)]')
```

and we bricked the problem as in the previous example

```
# model
md = gf.Model('real')

# add variable and data to model
md.add_fem_variable('u', mfu) # main unknown
md.add_initialized_fem_data('F', mfrhs, F) # volumic source term
md.add_initialized_fem_data('G', mfrhs, G) # Dirichlet condition
md.add_initialized_fem_data('H', mfrhs, H) # Neumann condition

# bricked the problem
md.add_Laplacian_brick(mim, 'u') # laplacian term on u
md.add_source_term_brick(mim, 'u', 'F') # volumic source term
md.add_normal_source_term_brick(mim, 'u', 'H', [down, right]) # Neumann condition

if (Dirichlet_with_multipliers): # Dirichlet condition on the top
    md.add_Dirichlet_condition_with_multipliers(mim, 'u', mfu, [top], 'G')
else:
    md.add_Dirichlet_condition_with_penalization(mim, 'u', dirichlet_coefficient, [top], 'G')

# Dirichlet condition on the right
# Two Dirichlet brick in order to test the multiplier selection in the intersection.
if (Dirichlet_with_multipliers):
    md.add_Dirichlet_condition_with_multipliers(mim, 'u', mfu, [right], 'G')
else:
    md.add_Dirichlet_condition_with_penalization(mim, 'u', dirichlet_coefficient, [right], 'G')
```

the only change is the add of *source term* brick. Finally the solution of the problem is extracted and exported

```
# mesh creation
m = gf.Mesh('regular_simplices', np.arange(0,1+1./NX,1./NX), np.arange(0,1+1./NX,1./NX))

# create a MeshFem for u and rhs fields of dimension 1 (i.e. a scalar field)
mfu = gf.MeshFem(m, 1)
mfrhs = gf.MeshFem(m, 1)
# assign the P2 fem to all convexes of the both MeshFem
mfu.set_fem(gf.Fem('FEM_PK(2,2)'))
mfrhs.set_fem(gf.Fem('FEM_PK(2,2)'))
```

view differences with gmsh sol.pos:

5.3 Final Laplacian with exact solution (Neumann boundary condition)

This example uses a mesh that was generated with [Gmsh](#), as in the *How-tos*. This example shows the basic usage of `getfem`, on the final canonical problem: solving the Laplacian, $-\Delta u = f$ on a square, with the Dirichlet condition

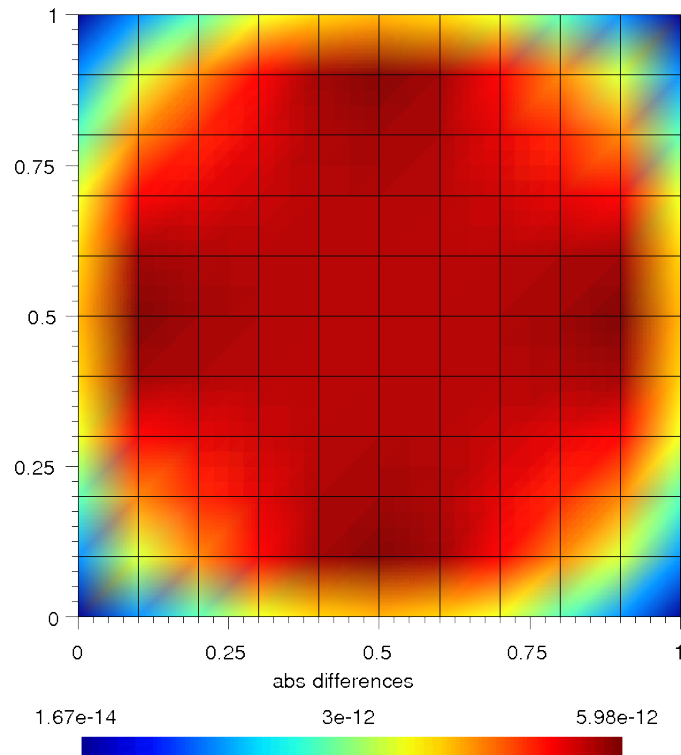


Figure 5.2: Differences

$u = g(x)$ on the domain boundary Γ_D and the Neumann condition $\frac{\partial u}{\partial \eta} = h(x)$ on the domain boundary Γ_N .

```

1  #!/usr/bin/env python
2  # -*- coding: UTF8 -*-
3
4  # import basic modules
5  import getfem as gf
6  import numpy as np
7
8  # boundary names
9  up    = 101 # m.region(101)
10 down  = 102 # m.region(102)
11 left  = 103 # m.region(103)
12 right = 104 # m.region(104)
13
14 # importing the mesh and boundary
15 m = gf.Mesh('import','gmsh','quad.msh')
16
17 # create a MeshFem of for a field of dimension 1 (i.e. a scalar field)
18 mf = gf.MeshFem(m,1)
19 # assign the P1 fem to all convexes of the MeshFem
20 mf.set_fem(gf.Fem('FEM_PK(2,1)'))
21
22 # an exact integration will be used
23 mim = gf.MeshIm(m,gf.Integ('IM_EXACT_SIMPLE(2)'))
24
25 # interpolate the exact solution
26 U = mf.eval('x[0]*(x[0]-1)*x[1]*(x[1]-1)')
27 # its Normal derivative on the domain boundary right (left is -DU)

```

```

28 DU = mf.eval(' (2*x[0]-1)*x[1]*(x[1]-1)')
29 # its laplacian
30 F = mf.eval(' -(2*(x[0]**2+x[1]**2)-2*(x[0]+x[1]))')
31
32 # generic elliptic brick
33 b0 = gf.MdBrick('generic_elliptic',mim,mf)
34 # add a Dirichlet condition on the domain boundary up
35 b1 = gf.MdBrick('dirichlet',b0,up,mf,'penalized')
36 b1.set_param('R',mf,U)
37 # add a Dirichlet condition on the domain boundary down
38 b2 = gf.MdBrick('dirichlet',b1,down,mf,'penalized')
39 b2.set_param('R',mf,U)
40 # add a source term
41 b3 = gf.MdBrick('source_term',b2)
42 b3.set_param('source_term',mf,F)
43 # add a Neumann condition on the domain boundary left
44 b4 = gf.MdBrick('source_term',b3,left)
45 b4.set_param('source_term',mf,-DU)
46 # add a Neumann condition on the domain boundary right
47 b5 = gf.MdBrick('source_term',b4,right)
48 b5.set_param('source_term',mf,DU)
49
50 # model state
51 mds = gf.MdState('real')
52 b5.solve(mds)
53
54 # extracted solution
55 csol = mds.get('state')
56 vsol = np.concatenate((0*csol,0*csol,csol))
57 vsol.shape=(3,-1)
58
59 # export
60 mf.export_to_pos('sol.pos',U,'Exact solution',
61                 csol,'Computed solution',
62                 vsol,'Displacement',
63                 abs(csol-U),'abs difference')

```

and view differences with `gmsht sol.pos`:

5.4 Linear and non-linear elasticity

This example uses a mesh that was generated with [GiD](#). The object is meshed with quadratic tetrahedrons. You can find the **py-file** of this example under the name **demo_tripod.py** in the directory `interface/tests/python/` of the *GetFEM++* distribution.

```

1 #This is the "modern" tripod demo, which uses the getfem model bricks
2
3 import getfem as gf
4 import numpy as np
5
6 # parameters
7 file_msh = 'tripod.GiD.msh'
8 degree = 2
9 linear = False
10 incompressible = False # ensure that degree > 1 when incompressible is on..

```

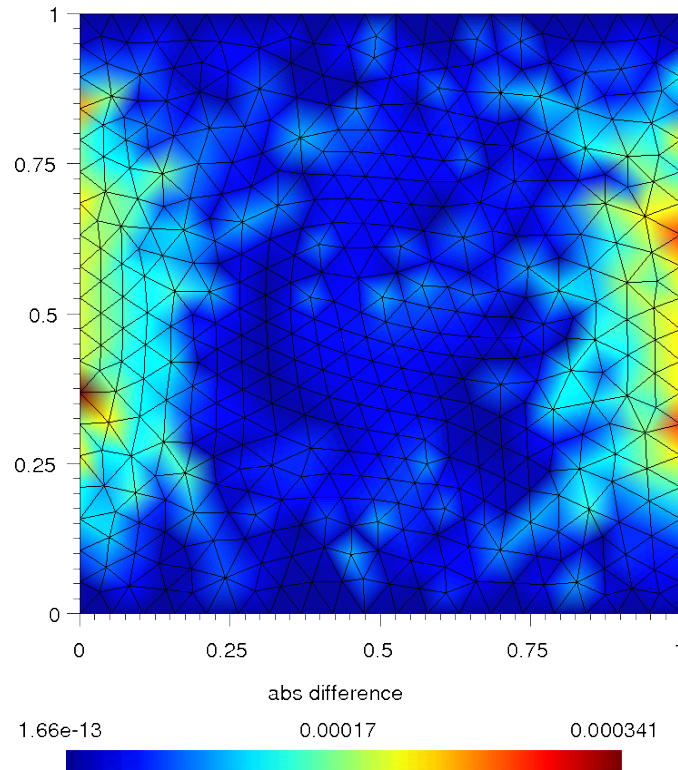


Figure 5.3: Differences

```

11 E = 1e3
12 Nu = 0.3
13 Lambda = E*Nu/((1+Nu)*(1-2*Nu))
14 Mu = E/(2*(1+Nu))
15
16 # create a Mesh object (importing)
17 m = gf.Mesh('import','gid',file_msh)
18 m.set('optimize_structure')
19
20 # create a MeshFem object
21 mfu = gf.MeshFem(m,3) # displacement
22 mfp = gf.MeshFem(m,1) # pressure
23 mfe = gf.MeshFem(m,3) # for plot displacement
24 mff = gf.MeshFem(m,1) # for plot von-mises
25 # assign the FEM
26 mfu.set_fem(gf.Fem('FEM_PK(3,%d)' % (degree,)))
27 mfp.set_fem(gf.Fem('FEM_PK_DISCONTINUOUS(3,0)'))
28 mfe.set_fem(gf.Fem('FEM_PK_DISCONTINUOUS(3,1,0.01)'))
29 mff.set_fem(gf.Fem('FEM_PK_DISCONTINUOUS(3,1,0.01)'))
30
31 # build a MeshIm object
32 mim = gf.MeshIm(m,gf.Integ('IM_TETRAHEDRON(5)'))
33
34 print 'nbcvs=%d, nbpts=%d, qdim=%d, fem = %s, nbdoef=%d' % \
35       (m.nbcvs(),m.nbpts(),mfu.qdim(),mfu.fem()[0].char(),mfu.nbdoef())
36
37 # detect some boundary of the mesh
38 P = m.pts()

```

```

39 ctop = (abs(P[1,:] - 13) < 1e-6)
40 cbot = (abs(P[1,:] + 10) < 1e-6)
41 pidtop = np.compress(ctop, range(0,m.nbpts()))
42 pidbot = np.compress(cbot, range(0,m.nbpts()))
43 ftop = m.faces_from_pid(pidtop)
44 fbot = m.faces_from_pid(pidbot)
45 # create boundary region
46 NEUMANN_BOUNDARY = 1
47 DIRICHLET_BOUNDARY = 2
48 m.set_region(NEUMANN_BOUNDARY, ftop)
49 m.set_region(DIRICHLET_BOUNDARY, fbot)
50
51 # the model bricks
52 if linear:
53     b0 = gf.MdBrick('isotropic_linearized_elasticity', mim, mfu)
54     b0.set_param('lambda', Lambda)
55     b0.set_param('mu', Mu)
56     if (incompressible):
57         b1 = gf.MdBrick('linear_incompressibility_term', b0, mfp)
58     else:
59         b1 = b0
60 else:
61     # large deformation with a linearized material law.. not a very good choice!
62     if (incompressible):
63         b0 = gf.MdBrick('nonlinear_elasticity', mim, mfu, 'Mooney_Rivlin')
64         b0.set_param('params', [Lambda, Mu])
65         b1 = gf.MdBrick('nonlinear_elasticity_incompressibility_term', b0, mfp)
66     else:
67         b0 = gf.MdBrick('nonlinear_elasticity', mim, mfu, 'SaintVenant_Kirchhoff')
68         #b0 = gf.MdBrick('nonlinear_elasticity', mim, mfu, 'Ciarlet_Geymonat')
69         b0.set_param('params', [Lambda, Mu])
70         b1 = b0
71
72 b2 = gf.MdBrick('source_term', b1, NEUMANN_BOUNDARY)
73 b2.set_param('source_term', [0, -10, 0])
74 b3 = gf.MdBrick('dirichlet', b2, DIRICHLET_BOUNDARY, mfu, 'penalized')
75
76 # create model state
77 mds = gf.MdState(b3)
78 # running solve...
79 b3.solve(mds, 'noisy', 'lsolver', 'superlu')
80
81 # extracted solution
82 U = mds.state()
83
84 # post-processing
85 VM = b0.von_mises(mds, mff)
86
87 # export U and VM in a pos file
88 sl = gf.Slice(('boundary',), mfu, 1)
89 sl.export_to_pos('sol.pos', mfu, U, 'Displacement', mff, VM, 'Von Mises Stress')
90
91 # save solution
92 #mfu.save('tripod.mf', 'with_mesh')
93 #U.tofile('tripod.U')
94 #mff.save('tripod.mff')
95 #VM.tofile('tripod.VM')
96 #gf.memstats()

```

Here is the final figure, displaying the Von Mises stress and displacements norms:

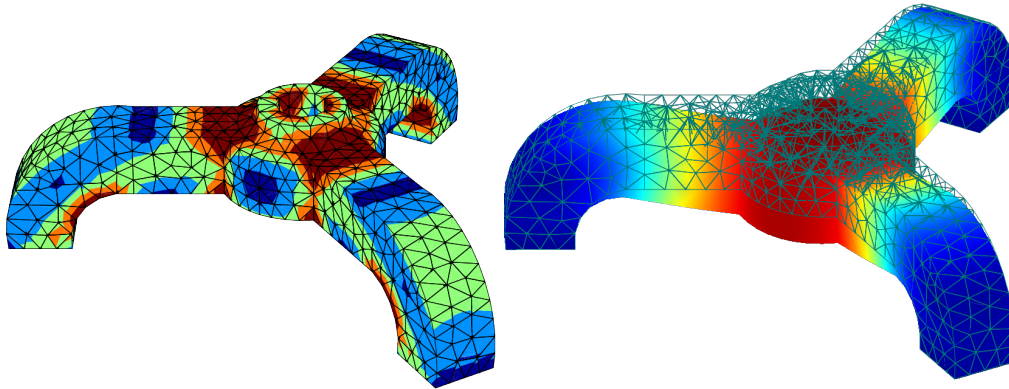


Figure 5.4: (a) Tripod Von Mises, (b) Tripod displacements norms.

5.5 Avoiding the bricks framework

The model bricks are very convenient, as they hide most of the details of the assembly of the final linear systems. However it is also possible to stay at a lower level, and handle the assembly of linear systems, and their resolution, directly in *Python*. For example, the demonstration **demo_tripod_alt.py** is very similar to the **demo_tripod.py** except that the assembly is explicit

```
# create a MeshFem object
mfd = gf.MeshFem(m,1) # data
# assign the FEM
mfd.set_fem(gf.Fem('FEM_PK(3,0)'))

# assembly
nbd = mfd.nbdof()
F = gf.asm_boundary_source(NEUMANN_BOUNDARY, mim, mfu, mfd, np.repeat([[0],[-100],[0]],nbd,1))
K = gf.asm_linear_elasticity(mim, mfu, mfd, np.repeat([Lambda], nbd), np.repeat([Mu], nbd))

# handle Dirichlet condition
(H,R) = gf.asm_dirichlet(DIRICHLET_BOUNDARY, mim, mfu, mfd, mfd.eval('numpy.identity(3)'), mfd.eval('1'))
(N,U0) = H.dirichlet_nullspace(R)

Nt = gf.Spmat('copy',N)
Nt.transpose()
KK = Nt*K*N
FF = Nt*F # FF = Nt*(F-K*U0)

# solve ...
P = gf.Precond('ildlt',KK)
UU = gf.linsolve_cg(KK,FF,P)
U = N*UU+U0

# post-processing
sl = gf.Slice(('boundary',), mfu, degree)
```

```

# compute the Von Mises Stress
DU = gf.compute_gradient(mfu,U,mfe)
VM = np.zeros((DU.shape[2],),'d')
Sigma = DU

for i in range(DU.shape[2]):
    d = np.array(DU[:, :, i])
    E = (d+d.T)*0.5
    Sigma[:, :, i]=E
    VM[i] = np.sum(E**2) - (1./3.)*np.sum(np.diagonal(E))**2

SigmaSL = gf.compute_interpolate_on(mfe,Sigma,sl)

# export to Gmsh
sl.export_to_pos('tripod.pos', mfe, VM,'Von Mises Stress', mfu, U, 'Displacement')
sl.export_to_pos('tripod_ev.pos', mfu, U, 'Displacement', SigmaSL, 'stress')

```

The Dirichlet condition $h(x)u(x) = r(x)$ is handled in the weak form $\int (h(x)u(x)).v(x) = \int r(x).v(x) \quad \forall v$ (where $h(x)$ is a 3×3 matrix field – here it is constant and equal to the identity). The reduced system $\mathbf{K}\mathbf{U} = \mathbf{F}$ is then built via the elimination of Dirichlet constraints from the original system. Note that it might be more efficient (and simpler) to deal with Dirichlet condition via a penalization technique.

5.6 Other examples

- the **demo_stokes_3D_tank.py** script shows a Stokes (viscous fluid) problem in a tank. The **demo_stokes_3D_tank_draw.py** shows how to draw a nice plot of the solution, with mesh slices and stream lines.
- the **demo_plasticity.py** script is an adaptation of the getfem++ example **tests/plasticity.cc**: a 2D or 3D bar is bended in many steps, and the plasticity of the material is taken into account (plastification occurs when the material's Von Mises exceeds a given threshold).

HOW-TOS

6.1 Import gmsh mesh

If we have in the file *quad.geo* a parameterized mesh, as this:

```
1 lc = 0.05 ;
2
3 Point(1) = {0,0,0,lc};
4 Point(2) = {1,0,0,lc};
5 Point(3) = {1,1,0,lc};
6 Point(4) = {0,1,0,lc};
7
8 Line(5) = {1,2};
9 Line(6) = {2,3};
10 Line(7) = {3,4};
11 Line(8) = {4,1};
12
13 Line Loop(9) = {5,6,7,8};
14 Plane Surface(10) = {9};
15
16 Physical Line(101) = {7};
17 Physical Line(102) = {5};
18 Physical Line(103) = {8};
19 Physical Line(104) = {6};
20
21 Physical Surface(201) = {10};
```

then, when we run:

```
$ gmsh -2 quad.geo
```

the file *quad.msh* is created and contains the encoding of the mesh and its regions. We can import that file (*quad.msh*) to getfem:

```
import getfem as gf

m = gf.Mesh('import','gmsh','quad.msh')
print m.regions()
```

with the second command we can see the *regions ids*. When we import the mesh, we might be warned with the following:

```
Level 3 Warning in getfem_import.cc, line 137:  
  All regions must have different number!
```

this means that the parametrization of the mesh in *Gmsh* *.geo* file must assign a **different** number to each region, the problem exists because in *Gmsh* can coexist, for example, “Physical Surface (200)” and “Physical Line (200)”, as they are different “types of regions” in *Gmsh*, that which does not occur in *GetFEM++* since there is only one “type of region”.

API REFERENCE

7.1 CvStruct

class CvStruct (*args)
GetFEM CvStruct object
General constructor for CvStruct objects

basic_structure ()
Get the simplest convex structure.
For example, the ‘basic structure’ of the 6-node triangle, is the canonical 3-noded triangle.

char ()
Output a string description of the CvStruct.

dim ()
Get the dimension of the convex structure.

display ()
displays a short summary for a CvStruct object.

face (*F*)
Return the convex structure of the face *F*.

facepts (*F*)
Return the list of point indices for the face *F*.

nbpts ()
Get the number of points of the convex structure.

7.2 Eltm

class Eltm (*args)
GetFEM Eltm object
This object represents a type of elementary matrix. In order to obtain a numerical value of theses matrices, see MeshIm.eltm().
If you have very particular assembling needs, or if you just want to check the content of an elementary matrix, this function might be useful. But the generic assembly abilities of gf_asm(...) should suit most needs.
General constructor for Eltm objects

- **E** = Eltm('base', Fem FEM) return a descriptor for the integration of shape functions on elements, using the Fem *FEM*.

- `E = Eltm('grad', Fem FEM)` return a descriptor for the integration of the gradient of shape functions on elements, using the Fem *FEM*.
- `E = Eltm('hessian', Fem FEM)` return a descriptor for the integration of the hessian of shape functions on elements, using the Fem *FEM*.
- `E = Eltm('normal')` return a descriptor for the unit normal of convex faces.
- `E = Eltm('grad_geotrans')` return a descriptor to the gradient matrix of the geometric transformation.
- `E = Eltm('grad_geotrans_inv')` return a descriptor to the inverse of the gradient matrix of the geometric transformation (this is rarely used).
- `E = Eltm('product', Eltm A, Eltm B)` return a descriptor for the integration of the tensorial product of elementary matrices *A* and *B*.

7.3 Fem

class Fem (*args)

GetFEM Fem object

This object represents a finite element method on a reference element.

General constructor for Fem objects

- `Fem TF = Fem('interpolated_fem', MeshFem mf, MeshIm mim, [ivec blocked_dof])` Build a special Fem which is interpolated from another MeshFem. Using this special finite element, it is possible to interpolate a given MeshFem *mf* on another mesh, given the integration method *mim* that will be used on this mesh. Note that this finite element may be quite slow, and eats much memory.
- `Fem(string fem_name)` The *fem_name* should contain a description of the finite element method. Please refer to the getfem++ manual (especially the description of finite element and integration methods) for a complete reference. Here is a list of some of them:
 - `FEM_PK(n,k)` classical Lagrange element Pk on a simplex of dimension *n*.
 - `FEM_PK_DISCONTINUOUS(N,K[,alpha])` discontinuous Lagrange element Pk on a simplex of dimension *n*.
 - `FEM_QK(n,k)` classical Lagrange element Qk on quadrangles, hexahedrons etc.
 - `FEM_QK_DISCONTINUOUS(n,k[,alpha])` discontinuous Lagrange element Qk on quadrangles, hexahedrons etc.
 - `FEM_Q2_INCOMPLETE` incomplete 2D Q2 element with 8 dof (serendipity Quad 8 element).
 - `FEM_PK_PRISM(n,k)` classical Lagrange element Pk on a prism.
 - `FEM_PK_PRISM_DISCONTINUOUS(n,k[,alpha])` classical discontinuous Lagrange element Pk on a prism.
 - `FEM_PK_WITH_CUBIC_BUBBLE(n,k)` classical Lagrange element Pk on a simplex with an additional volumic bubble function.
 - `FEM_P1_NONCONFORMING` non-conforming P1 method on a triangle.
 - `FEM_P1_BUBBLE_FACE(n)` P1 method on a simplex with an additional bubble function on face 0.
 - `FEM_P1_BUBBLE_FACE_LAG` P1 method on a simplex with an additional lagrange dof on face 0.
 - `FEM_PK_HIERARCHICAL(n,k)` PK element with a hierarchical basis.
 - `FEM_QK_HIERARCHICAL(n,k)` QK element with a hierarchical basis
 - `FEM_PK_PRISM_HIERARCHICAL(n,k)` PK element on a prism with a hierarchical basis.
 - `FEM_STRUCTURED_COMPOSITE(FEM,k)` Composite fem on a grid with *k* divisions.
 - `FEM_PK_HIERARCHICAL_COMPOSITE(n,k,s)` Pk composite element on a grid with *s* subdivisions and with a hierarchical basis.

- FEM_PK_FULL_HIERARCHICAL_COMPOSITE(n,k,s) Pk composite element with s subdivisions and a hierarchical basis on both degree and subdivision.
- FEM_PRODUCT(FEM1,FEM2) tensorial product of two polynomial elements.
- FEM_HERMITE(n) Hermite element P3 on a simplex of dimension $n = 1, 2, 3$.
- FEM_ARGYRIS Argyris element P5 on the triangle.
- FEM_HCT_TRIANGLE Hsieh-Clough-Tocher element on the triangle (composite P3 element which is C^1), should be used with IM_HCT_COMPOSITE() integration method.
- FEM_QUADC1_COMPOSITE Quadrilateral element, composite P3 element and C^1 (16 dof).
- FEM_REDUCED_QUADC1_COMPOSITE Quadrilateral element, composite P3 element and C^1 (12 dof).
- FEM_RT0(n) Raviart-Thomas element of order 0 on a simplex of dimension n .
- FEM_NEDELEC(n) Nedelec edge element of order 0 on a simplex of dimension n .

Of course, you have to ensure that the selected fem is compatible with the geometric transformation: a Pk fem has no meaning on a quadrangle.

base_value (p)

Evaluate all basis functions of the FEM at point p .
 p is supposed to be in the reference convex!

char ()

Output a (unique) string representation of the Fem.
 This can be used to perform comparisons between two different Fem objects.

dim ()

Return the dimension (dimension of the reference convex) of the Fem.

display ()

displays a short summary for a Fem object.

estimated_degree ()

Return an estimation of the polynomial degree of the Fem.
 This is an estimation for fem which are not polynomials.

grad_base_value (p)

Evaluate the gradient of all base functions of the Fem at point p .
 p is supposed to be in the reference convex!

hess_base_value (p)

Evaluate the Hessian of all base functions of the Fem at point p .
 p is supposed to be in the reference convex!.

is_equivalent ()

Return 0 if the Fem is not equivalent.
 Equivalent Fem are evaluated on the reference convex. This is the case of most classical Fem's.

is_lagrange ()

Return 0 if the Fem is not of Lagrange type.

is_polynomial ()

Return 0 if the basis functions are not polynomials.

nbdof ($cv=None$)

Return the number of dof for the Fem.
 Some specific Fem (for example 'interpolated_fem') may require a convex number cv to give their result.
 In most of the case, you can omit this convex number.

poly_str ()

Return the polynomial expressions of its basis functions in the reference convex.
 The result is expressed as a tuple of strings. Of course this will fail on non-polynomial Fem's.

pts (*cv=None*)

Get the location of the dof on the reference element.

Some specific Fem may require a convex number *cv* to give their result (for example 'interpolated_fem').
In most of the case, you can omit this convex number.

target_dim ()

Return the dimension of the target space.

The target space dimension is usually 1, except for vector Fem.

7.4 GeoTrans

class GeoTrans (*args)

GetFEM GeoTrans object

The geometric transformation must be used when you are building a custom mesh convex by convex (see the `add_convex()` function of `Mesh`): it also defines the kind of convex (triangle, hexahedron, prism, etc..)

General constructor for GeoTrans objects

• `GeoTrans = GeoTrans(string name)` The name argument contains the specification of the geometric transformation as a string, which may be:

– `GT_PK(n,k)` Transformation on simplexes, dim *n*, degree *k*.

– `GT_QK(n,k)` Transformation on parallelepipeds, dim *n*, degree *k*.

– `GT_PRISM(n,k)` Transformation on prisms, dim *n*, degree *k*.

– `GT_PRODUCT(A,B)` Tensorial product of two transformations.

– `GT_LINEAR_PRODUCT(A,B)` Linear tensorial product of two transformations

char ()

Output a (unique) string representation of the GeoTrans.

This can be used to perform comparisons between two different GeoTrans objects.

dim ()

Get the dimension of the GeoTrans.

This is the dimension of the source space, i.e. the dimension of the reference convex.

display ()

displays a short summary for a GeoTrans object.

is_linear ()

Return 0 if the GeoTrans is not linear.

nbpts ()

Return the number of points of the GeoTrans.

normals ()

Get the normals for each face of the reference convex of the GeoTrans.

The normals are stored in the columns of the output matrix.

pts ()

Return the reference convex points of the GeoTrans.

The points are stored in the columns of the output matrix.

transform (*G, Pr*)

Apply the GeoTrans to a set of points.

G is the set of vertices of the real convex, *Pr* is the set of points (in the reference convex) that are to be transformed. The corresponding set of points in the real convex is returned.

7.5 GlobalFunction

class GlobalFunction (*args)

GetFEM GlobalFunction object

Global function object is represented by three functions:

- The global function *val*.
- The global function gradient *grad*.
- The global function Hessian *hess*.

this type of function is used as local and global enrichment function. The global function Hessian is an optional parameter (only for fourth order derivative problems).

General constructor for GlobalFunction objects

- GF = GlobalFunction('cutoff', int fn, scalar r, scalar r1, scalar r0)
Create a cutoff global function.
- GF = GlobalFunction('crack', int fn) Create a near-tip asymptotic global function for modelling cracks.
- GF = GlobalFunction('parser', string val[, string grad[, string hess]]) Create a global function from strings *val*, *grad* and *hess*.
- GF = GlobalFunction('product', GlobalFunction F, GlobalFunction G) Create a product of two global functions.
- GF = GlobalFunction('add', GlobalFunction F, GlobalFunction G) Create a add of two global functions.

char ()

Output a (unique) string representation of the GlobalFunction.

This can be used to perform comparisons between two different GlobalFunction objects. This function is to be completed.

display ()

displays a short summary for a GlobalFunction object.

grad (PTs)

Return *grad* function evaluation in *PTs* (column points).

On return, each column of *GRADs* is of the form [Gx,Gy].

hess (PTs)

Return *hess* function evaluation in *PTs* (column points).

On return, each column of *HESSs* is of the form [Hxx,Hxy,Hyx,Hyy].

val (PTs)

Return *val* function evaluation in *PTs* (column points).

7.6 Integ

class Integ (*args)

GetFEM Integ object

General object for obtaining handles to various integrations methods on convexes (used when the elementary matrices are built).

General constructor for Integ objects

•`Integ(string method)` Here is a list of some integration methods defined in `getfem++` (see the description of finite element and integration methods for a complete reference):

- `IM_EXACT_SIMPLEX(n)` Exact integration on simplices (works only with linear geometric transformations and PK Fem's).
- `IM_PRODUCT(A,B)` Product of two integration methods.
- `IM_EXACT_PARALLELEPIPED(n)` Exact integration on parallelepipeds.
- `IM_EXACT_PRISM(n)` Exact integration on prisms.
- `IM_GAUSS1D(k)` Gauss method on the segment, order $k=1,3,\dots,99$.
- `IM_NC(n,k)` Newton-Cotes approximative integration on simplexes, order k .
- `IM_NC_PARALLELEPIPED(n,k)` Product of Newton-Cotes integration on parallelepipeds.
- `IM_NC_PRISM(n,k)` Product of Newton-Cotes integration on prisms.
- `IM_GAUSS_PARALLELEPIPED(n,k)` Product of Gauss1D integration on parallelepipeds.
- `IM_TRIANGLE(k)` Gauss methods on triangles $k=1,3,5,6,7,8,9,10,13,17,19$.
- `IM_QUAD(k)` Gauss methods on quadrilaterons $k=2, 3, 5, \dots, 17$. Note that `IM_GAUSS_PARALLELEPIPED` should be preferred for QK Fem's.
- `IM_TETRAHEDRON(k)` Gauss methods on tetrahedrons $k=1, 2, 3, 5, 6$ or 8 .
- `IM_SIMPLEX4D(3)` Gauss method on a 4-dimensional simplex.
- `IM_STRUCTURED_COMPOSITE(im,k)` Composite method on a grid with k divisions.
- `IM_HCT_COMPOSITE(im)` Composite integration suited to the HCT composite finite element.

Example:

```
–Integ('IM_PRODUCT(IM_GAUSS1D(5),IM_GAUSS1D(5))')
```

is the same as:

```
–Integ('IM_GAUSS_PARALLELEPIPED(2,5)')
```

Note that 'exact integration' should be avoided in general, since they only apply to linear geometric transformations, are quite slow, and subject to numerical stability problems for high degree Fem's.

char()

Output a (unique) string representation of the integration method.

This can be used to comparisons between two different `Integ` objects.

coeffs()

Returns the coefficients associated to each integration point.

Only for approximate methods, this has no meaning for exact integration methods!

dim()

Return the dimension of the reference convex of the method.

display()

displays a short summary for a `Integ` object.

face_coeffs(F)

Returns the coefficients associated to each integration of a face.

Only for approximate methods, this has no meaning for exact integration methods!

face_pts(F)

Return the list of integration points for a face.

Only for approximate methods, this has no meaning for exact integration methods!

is_exact()

Return 0 if the integration is an approximate one.

nbpts()

Return the total number of integration points.

Count the points for the volume integration, and points for surface integration on each face of the reference convex.<Par>

Only for approximate methods, this has no meaning for exact integration methods!

pts()

Return the list of integration points

Only for approximate methods, this has no meaning for exact integration methods!

7.7 LevelSet

class LevelSet (*args)

GetFEM LevelSet object

The level-set object is represented by a primary level-set and optionally a secondary level-set used to represent fractures (if $p(x)$ is the primary level-set function and $s(x)$ is the secondary level-set, the crack is defined by $p(x)=0$ and $s(x)\leq 0$: the role of the secondary is to determine the crack front/tip).

IMPORTANT: All tools listed below need the package qhull installed on your system. This package is widely available. It computes convex hull and delaunay triangulations in arbitrary dimension.

General constructor for LevelSet objects

```
•LS = LevelSet(Mesh m, int d[, string 'ws' | string func_1[, string
  func_2 | string 'ws']])
```

Create a LevelSet object on a Mesh represented by a primary function (and optional secondary function, both) defined on a lagrange MeshFem of degree d . If *ws* (with secondary) is set; this levelset is represented by a primary function and a secondary function. If *func_1* is set; the primary function is defined by that expression. If *func_2* is set; this levelset is represented by a primary function and a secondary function defined by these expressions.

char()

Output a (unique) string representation of the LevelSet.

This can be used to perform comparisons between two different LevelSet objects. This function is to be completed.

degree()

Return the degree of lagrange representation.

display()

displays a short summary for a LevelSet.

memsize()

Return the amount of memory (in bytes) used by the level-set.

mf()

Return a reference on the MeshFem object.

set_values (*args)

Synopsis: LevelSet.set_values(self, {mat v1|string func_1}[, mat v2|string func_2])

Set values of the vector of dof for the level-set functions.

Set the primary function with the vector of dof $v1$ (or the expression *func_1*) and the secondary function (if any) with the vector of dof $v2$ (or the expression *func_2*)

simplify (eps=0.01)

Simplify dof of level-set optionally with the parameter *eps*.

values (nls)

Return the vector of dof for *nls* funtion.

If *nls* is 0, the method return the vector of dof for the primary level-set funtion. If *nls* is 1, the method return the vector of dof for the secondary level-set function (if any).

7.8 MdBrick

class MdBrick (*args)

GetFEM MdBrick object

General constructor for MdBrick objects

• `B = MdBrick('constraint', MdBrick pb, string CTYPE[, int nfem])` Build a generic constraint brick.

It may be useful in some situations, such as the Stokes problem where the pressure is defined modulo a constant. In such a situation, this brick can be used to add an additional constraint on the pressure value. *CTYPE* has to be chosen among 'augmented', 'penalized', and 'eliminated'. The constraint can be specified with `MdBrick.constraints()`. Note that Dirichlet bricks (except the 'generalized Dirichlet' one) are also specializations of the 'constraint' brick.

• `B = MdBrick('dirichlet', MdBrick pb, int bnum, MeshFem mf_m, string CTYPE[, int nfem])` Build a Dirichlet condition brick which impose the value of a field along a mesh boundary.

The *bnum* parameter selects on which mesh region the Dirichlet condition is imposed. *CTYPE* has to be chosen among 'augmented', 'penalized', and 'eliminated'. The *mf_m* may generally be taken as the MeshFem of the unknown, but for 'augmented' Dirichlet conditions, you may have to respect the Inf-Sup condition and choose an adequate MeshFem.

• `B = MdBrick('dirichlet on normal component', MdBrick pb, int bnum, MeshFem mf_m, string CTYPE[, int nfem])` Build a Dirichlet condition brick which imposes the value of the normal component of a vector field.

• `B = MdBrick('dirichlet on normal derivative', MdBrick pb, int bnum, MeshFem mf_m, string CTYPE[, int nfem])` Build a Dirichlet condition brick which imposes the value of the normal derivative of the unknown.

• `B = MdBrick('generalized dirichlet', MdBrick pb, int bnum[, int nfem])`
This is the "old" Dirichlet brick of getfem.

This brick can be used to impose general Dirichlet conditions $h(x)u(x) = r(x)$, however it may have some issues with elaborated Fem's (such as Argyris, etc). It should be avoided when possible.

• `B = MdBrick('source term', MdBrick pb[, int bnum=-1[, int nfem]])` Add a boundary or volumic source term (int *B.v*).

If *bnum* is omitted (or set to -1), the brick adds a volumic source term on the whole mesh. For *bnum* ≥ 0 , the source term is imposed on the mesh region *bnum*. Use `MdBrick.set_param('source term',mf,B)` to set the source term field. The source term is expected as a vector field of size *Q* (with *Q* = qdim).

• `B = MdBrick('normal source term', MdBrick pb, int bnum[, int nfem])` Add a boundary source term (int (*Bn.v*)).

The source term is imposed on the mesh region *bnum* (which of course is not allowed to be a volumic region, only boundary regions are allowed). Use `MdBrick.set_param('source term',mf,B)` to set the source term field. The source term *B* is expected as tensor field of size *Q* $\times$ *N* (with *Q* = qdim, *N* = mesh dim). For example, if you consider an elasticity problem, this brick may be used to impose a force on the boundary with *B* as the stress tensor.

• `B = MdBrick('normal derivative source term', MdBrick parent, int bnum[, int nfem])` Add a boundary source term (int (partial_n *B*).v).

The source term is imposed on the mesh region *bnum*. Use `MdBrick.set_param('source term',mf,B)` to set the source term field, which is expected as a vector field of size *Q* (with *Q* = qdim).

• `B = MdBrick('neumann KirchhoffLove source term', MdBrick pb, int bnum[, int nfem])` Add a boundary source term for neumann Kirchhoff-Love plate problems.
Should be used with the Kirchhoff-Love flavour of the bilaplacian brick.

• `B = MdBrick('qu term', MdBrick pb[, int bnum[, int nfem]])` Update the tangent matrix with a int (Qu).v term.

The `Q(x)` parameter is a matrix field of size `qdim x qdim`. An example of use is for the “iku” part of Robin boundary conditions `partial_n u + iku = ...`

• `B = MdBrick('mass matrix', MeshIm mim, MeshFem mf_u[, 'real' | 'complex'])` Build a mass-matrix brick.

• `B = MdBrick('generic elliptic', MeshIm mim, MeshFem mfu[, 'scalar' | 'matrix' | 'tensor'][, 'real' | 'complex'])` Setup a generic elliptic problem.
`a(x)*grad(U).grad(V)`

The brick parameter *a* may be a scalar field, a matrix field, or a tensor field (default is scalar).

• `B = MdBrick('helmholtz', MeshIm mim, MeshFem mfu[, 'real' | 'complex'])`
 Setup a Helmholtz problem.

The brick has one parameter, ‘wave_number’.

• `B = MdBrick('isotropic linearized elasticity', MeshIm mim, MeshFem mfu)` Setup a linear elasticity problem.

The brick has two scalar parameter, ‘lambda’ and ‘mu’ (the Lamé coefficients).

• `B = MdBrick('linear incompressibility term', MdBrick pb, MeshFem mfp[, int nfem])` Add an incompressibility constraint ($\text{div } \mathbf{u} = 0$).

• `B = MdBrick('nonlinear elasticity', MeshIm mim, MeshFem mfu, string law)` Setup a nonlinear elasticity (large deformations) problem.

The material law can be chosen among: – ‘Saint Venant Kirchhoff’ Linearized material law.

– ‘Mooney Rivlin’ To be used with the nonlinear incompressibility term.

– ‘Ciarlet Geymonat’

• `B = MdBrick('nonlinear elasticity incompressibility term', MdBrick pb, MeshFem mfp[, int nfem])` Add an incompressibility constraint to a large strain elasticity problem.

• `B = MdBrick('small deformations plasticity', MeshIm mim, MeshFem mfu, scalar THRESHOLD)` Setup a plasticity problem (with small deformations).

The *THRESHOLD* parameter is the maximum value of the Von Mises stress before ‘plastification’ of the material.

• `B = MdBrick('dynamic', MdBrick pb, scalar rho[, int numfem])` Dynamic brick. This brick is not fully working.

• `B = MdBrick('bilaplacian', MeshIm mim, MeshFem mfu[, 'Kirchhoff-Love'])` Setup a bilaplacian problem.

If the ‘Kirchhoff-Love’ option is specified, the Kirchhoff-Love plate model is used.

• `B = MdBrick('navier stokes', MeshIm mim, MeshFem mfu, MeshFem mfp)` Setup a Navier-Stokes problem (this brick is not ready, do not use it).

• `B = MdBrick('isotropic_linearized_plate', MeshIm mim, MeshIm mims, MeshFem mfut, MeshFem mfu3, MeshFem mftheta, scalar eps)` Setup a linear plate model brick.

For moderately thick plates, using the Reissner-Mindlin model. *eps* is the plate thickness, the MeshFem *mfut* and *mfu3* are used respectively for the membrane displacement and the transverse displacement of the plate. The MeshFem *mftheta* is the rotation of the normal (“section rotations”).

The second integration method *mims* can be chosen equal to *mim*, or different if you want to perform sub-integration on the transverse shear term (mitc4 projection).

This brick has two parameters “lambda” and “mu” (the Lamé coefficients)

• `B = MdBrick('mixed_isotropic_linearized_plate', MeshIm mim, MeshFem mfut, MeshFem mfu3, MeshFem mftheta, scalar eps)` Setup a mixed linear plate model brick.

For thin plates, using Kirchhoff-Love model. For a non-mixed version, use the bilaplacian brick.

`•B = MdBrick('plate_source_term', MdBrick pb[, int bnum=-1[, int nfem]])` Add a boundary or a volumic source term to a plate problem.

This brick has two parameters: “B” is the displacement (ut and u3) source term, “M” is the moment source term (i.e. the source term on the rotation of the normal).

`•B = MdBrick('plate_simple_support', MdBrick pb, int bnum, string CTYPE[, int nfem])` Add a “simple support” boundary condition to a plate problem.

Homogeneous Dirichlet condition on the displacement, free rotation. *CTYPE* specifies how the constraint is enforced (‘penalized’, ‘augmented’ or ‘eliminated’).

`•B = MdBrick('plate_clamped_support', MdBrick pb, int bnum, string CTYPE[, int nfem])` Add a “clamped support” boundary condition to a plate problem.

Homogeneous Dirichlet condition on the displacement and on the rotation. *CTYPE* specifies how the constraint is enforced (‘penalized’, ‘augmented’ or ‘eliminated’).

`•B = MdBrick('plate_closing', MdBrick pb[, int nfem])` Add a free edges condition for the mixed plate model brick.

This brick is required when the mixed linearized plate brick is used. It must be inserted after all other boundary conditions (the reason is that the brick has to inspect all other boundary conditions to determine the number of disconnected boundary parts which are free edges).

char()

Output a (unique) string representation of the MdBrick.

This can be used to perform comparisons between two different MdBrick objects. This function is to be completed.

constraints(H, R)

Set the constraints imposed by a constraint brick.

This is only applicable to the bricks which inherit from the constraint brick, such as the Dirichlet ones. Imposes $H.U=R$.

constraints_rhs(H, R)

Set the right hand side of the constraints imposed by a constraint brick.

This is only applicable to the bricks which inherit from the constraint brick, such as the Dirichlet ones.

dim()

Get the dimension of the main mesh (2 for a 2D mesh, etc).

display()

displays a short summary for a MdBrick.

is_coercive()

Return true if the problem is coercive.

is_complex()

Return true if the problem uses complex numbers.

is_linear()

Return true if the problem is linear.

is_symmetric()

Return true if the problem is symmetric.

memsize()

Return the amount of memory (in bytes) used by the model brick.

mixed_variables()

Identify the indices of mixed variables (typically the pressure, etc.) in the tangent matrix.

nb_constraints()

Get the total number of dof constraints of the current problem.

This is the sum of the brick specific dof constraints plus the dof constraints of the parent bricks.

nbdof ()

Get the total number of dof of the current problem.

This is the sum of the brick specific dof plus the dof of the parent bricks.

param (parameter_name)

Get the parameter value.

When the parameter has been assigned a specific MeshFem, it is returned as a large array (the last dimension being the MeshFem dof). When no MeshFem has been assigned, the parameter is considered to be constant over the mesh.

param_list ()

Get the list of parameters names.

Each brick embeds a number of parameters (the Lamé coefficients for the linearized elasticity brick, the wave number for the Helmholtz brick,...), described as a (scalar, or vector, tensor etc) field on a mesh_fem.

You can read/change the parameter values with MdBrick.param() and MdBrick.param().

penalization_epsilon (eps)

Change the penalization coefficient of a constraint brick.

This is only applicable to the bricks which inherit from the constraint brick, such as the Dirichlet ones. And of course it is not effective when the constraint is enforced via direct elimination or via Lagrange multipliers. The default value of *eps* is 1e-9.

set_param (name, *args)

Synopsis: MdBrick.set_param(self, string name, {MeshFem mf, V | V })

Change the value of a brick parameter.

name is the name of the parameter. *V* should contain the new parameter value (vector or float). If a MeshFem is given, *V* should hold the field values over that MeshFem (i.e. its last dimension should be MeshFem.nbdof() or 1 for constant field).

solve (mds, *args)

Synopsis: MdBrick.solve(self, MdState mds[,...])

Run the standard getfem solver.

Note that you should be able to use your own solver if you want (it is possible to obtain the tangent matrix and its right hand side with the MdState.tangent_matrix() etc.).

Various options can be specified:

- **‘noisy’ or ‘very noisy’** the solver will display some information showing the progress (residual values etc.).
- **‘max_iter’, NIT** set the maximum iterations numbers.
- **‘max_res’, RES** set the target residual value.
- **‘lsolver’, SOLVERNAME** select explicitly the solver used for the linear systems (the default value is ‘auto’, which lets getfem choose itself). Possible values are ‘superlu’, ‘mumps’ (if supported), ‘cg/ildlt’, ‘gmres/ilu’ and ‘gmres/ilut’.

subclass ()

Get the typename of the brick.

tresca (mds, mft)

Compute the Tresca stress criterion on the MeshFem *mft*.

Only available on bricks where it has a meaning: linearized elasticity, plasticity, nonlinear elasticity.

von_mises (mds, mfvf)

Compute the Von Mises stress on the MeshFem *mfvf*.

Only available on bricks where it has a meaning: linearized elasticity, plasticity, nonlinear elasticity. Note that in 2D it is not the “real” Von Mises (which should take into account the ‘plane stress’ or ‘plane strain’ aspect), but a pure 2D Von Mises.

7.9 MdState

class MdState (*args)

GetFEM MdState object

A model state is an object which store the state data for a chain of model bricks. This includes the global tangent matrix, the right hand side and the constraints.

This object is now deprecated and replaced by the Model object.

There are two sorts of model states, the *real* and the *complex* models states.

General constructor for MdState objects

- **MDS** = **MdState**('real') Build a model state for real unknowns.
- **MDS** = **MdState**('complex') Build a model state for complex unknowns.
- **MDS** = **MdState**(**MdBrick** B) Build a modelstate for the brick *B*.
Selects the real or complex state from the complexity of *B*.

char()

Output a (unique) string representation of the MdState.

This can be used to perform comparisons between two different MdState objects. This function is to be completed.

clear()

Clear the model state.

compute_reduced_residual()

Compute the reduced residual from the residual and constraints.

compute_reduced_system()

Compute the reduced system from the tangent matrix and constraints.

compute_residual(*B*)

Compute the residual for the brick *B*.

compute_tangent_matrix(*B*)

Update the tangent matrix from the brick *B*.

constraints_matrix()

Return the constraints matrix stored in the model state.

constraints_nullspace()

Return the nullspace of the constraints matrix.

display()

displays a short summary for a MdState.

is_complex()

Return 0 if the model state is real, 1 if it is complex.

memsize()

Return the amount of memory (in bytes) used by the model state.

reduced_residual()

Return the residual on the reduced system.

reduced_tangent_matrix()

Return the reduced tangent matrix (i.e. the tangent matrix after elimination of the constraints).

residual()

Return the residual.

set_state(*U*)

Update the internal state with the vector *U*.

state()

Return the vector of unknowns, which contains the solution after `MdBrick.solve()`.

tangent_matrix()

Return the tangent matrix stored in the model state.

unreduce(*U*)

Reinsert the constraint eliminated from the system.

7.10 Mesh

class Mesh(*args)

GetFEM Mesh object

This object is able to store any element in any dimension even if you mix elements with different dimensions.

General constructor for Mesh objects

- `M = Mesh('empty', int dim)` Create a new empty mesh.
- `M = Mesh('cartesian', vec X[, vec Y[, vec Z, ...]])` Build quickly a regular mesh of quadrangles, cubes, etc.
- `M = Mesh('triangles grid', vec X, vec Y)` Build quickly a regular mesh of triangles.
This is a very limited and somehow deprecated function (See also `Mesh('ptND')`, `Mesh('regular simplices')` and `Mesh('cartesian')`).
- `M = Mesh('regular simplices', vec X[, vec Y[, vec Z, ...]]['degree', int k][ 'noised' ])` Mesh a n-dimensionnal parallelepiped with simplices (triangles, tetrahedrons etc).
The optional degree may be used to build meshes with non linear geometric transformations.
- `M = Mesh('curved', Mesh m0, vec F)` Build a curved (n+1)-dimensions mesh from a n-dimensions mesh *m0*.
The points of the new mesh have one additional coordinate, given by the vector *F*. This can be used to obtain meshes for shells. *m0* may be a MeshFem object, in that case its linked mesh will be used.
- `M = Mesh('prismatic', Mesh m0, int NLAY)` Extrude a prismatic Mesh *M* from a Mesh *m0*.
In the additional dimension there are *NLAY* layers of elements built from 0 to 1.
- `M = Mesh('pt2D', mat P, ivec T[, int n])` Build a mesh from a 2D triangulation.
Each column of *P* contains a point coordinate, and each column of *T* contains the point indices of a triangle. *n* is optional and is a zone number. If *n* is specified then only the zone number *n* is converted (in that case, *T* is expected to have 4 rows, the fourth containing these zone numbers).
- `M = Mesh('ptND', mat P, imat T)` Build a mesh from a N-dimensional “triangulation”.
Similar function to ‘pt2D’, for building simplex meshes from a triangulation given in *T*, and a list of points given in *P*. The dimension of the mesh will be the number of rows of *P*, and the dimension of the simplex will be the number of rows of *T*.
- `M = Mesh('load', string filename)` Load a mesh from a GETFEM++ ascii mesh file.
See also `Mesh.save(string filename)`.
- `M = Mesh('from string', string s)` Load a mesh from a string description.
For example, a string returned by `Mesh.char()`.
- `M = Mesh('import', string format, string filename)` Import a mesh.
format may be:
 - ‘gmsb’ for a mesh created with *Gmsb*
 - ‘gid’ for a mesh created with *GiD*

–‘am_fmt’ for a mesh created with *EMC2*

•**M = Mesh(‘clone’, Mesh m2)** Create a copy of a mesh.

add_convex (*GT, PTS*)

Add a new convex into the mesh.

The convex structure (triangle, prism,...) is given by *GT* (obtained with *GeoTrans*(‘...’)), and its points are given by the columns of *PTS*. On return, *CVIDs* contains the convex #ids. *PTS* might be a 3-dimensional array in order to insert more than one convex (or a two dimensional array correctly shaped according to Fortran ordering).

add_point (*PTS*)

Insert new points in the mesh and return their #ids.

PTS should be an *n* × *m* matrix, where *n* is the mesh dimension, and *m* is the number of points that will be added to the mesh. On output, *PIDs* contains the point #ids of these new points.

Remark: if some points are already part of the mesh (with a small tolerance of approximately $1e-8$), they won’t be inserted again, and *PIDs* will contain the previously assigned #ids of these points.

boundaries ()

DEPRECATED FUNCTION. Use ‘regions’ instead.

boundary ()

DEPRECATED FUNCTION. Use ‘region’ instead.

char ()

Output a string description of the mesh.

convex_area (*CVIDs=None*)

Return an estimation of the area of each convex.

curved_edges (*N, CVLST=None*)

[OBSOLETE FUNCTION! will be removed in a future release]

More sophisticated version of *Mesh.edges()* designed for curved elements. This one will return *N* (*N* ≥ 2) points of the (curved) edges. With *N* = 2, this is equivalent to *Mesh.edges()*. Since the points are no more always part of the mesh, their coordinates are returned instead of points number, in the array *E* which is a [*mesh_dim* × 2 × *nb_edges*] array. If the optional output argument *C* is specified, it will contain the convex number associated with each edge.

cvid ()

Return the list of all convex #id.

Note that their numbering is not supposed to be contiguous from 0 to *Mesh.nbcv*s()–1, especially if some points have been removed from the mesh. You can use *Mesh.optimize_structure()* to enforce a contiguous numbering.

cvid_from_pid (*PIDs, share=False*)

Search convex #ids related with the point #ids given in *PIDs*.

If *share=False*, search convex whose vertex #ids are in *PIDs*. If *share=True*, search convex #ids that share the point #ids given in *PIDs*. *CVIDs* is a vector (possibly empty).

cvstruct (*CVIDs=None*)

Return an array of the convex structures.

If *CVIDs* is not given, all convexes are considered. Each convex structure is listed once in *S*, and *CV2S* maps the convexes indice in *CVIDs* to the indice of its structure in *S*.

del_convex (*CVIDs*)

Remove one or more convexes from the mesh.

CVIDs should contain the convexes #ids, such as the ones returned by the ‘add convex’ command.

del_convex_of_dim (*DIMs*)

Remove all convexes of dimension listed in *DIMs*.

For example; *Mesh.del_convex_of_dim*([1, 2]) remove all line segments, triangles and quadrangles.

del_point (*PIDs*)

Removes one or more points from the mesh.

PIDs should contain the point #ids, such as the one returned by the 'add point' command.

delete_boundary (*rnum*, *CVFIDs*)

DEPRECATED FUNCTION. Use 'delete region' instead.

delete_region (*RIDs*)

Remove the regions whose #ids are listed in *RIDs*

dim ()

Get the dimension of the mesh (2 for a 2D mesh, etc).

display ()

displays a short summary for a Mesh object.

edges (*CVLST=None*, **args*)

Synopsis: [E,C] = Mesh.edges(self [, CVLST][, 'merge'])

[OBSOLETE FUNCTION! will be removed in a future release]

Return the list of edges of mesh M for the convexes listed in the row vector CVLST. E is a 2 x nb_edges matrix containing point indices. If CVLST is omitted, then the edges of all convexes are returned. If CVLST has two rows then the first row is supposed to contain convex numbers, and the second face numbers, of which the edges will be returned. If 'merge' is indicated, all common edges of convexes are merged in a single edge. If the optional output argument C is specified, it will contain the convex number associated with each edge.

export_to_dx (*filename*, **args*)

Synopsis: Mesh.export_to_dx(self, string filename, ... [, 'ascii'][, 'append'][, 'as', string name[, 'serie', string serie_name]][, 'edges'])

Exports a mesh to an OpenDX file.

See also MeshFem.export_to_dx(), Slice.export_to_dx().

export_to_pos (*filename*, *name=None*)

Exports a mesh to a POS file .

See also MeshFem.export_to_pos(), Slice.export_to_pos().

export_to_vtk (*filename*, **args*)

Synopsis: Mesh.export_to_vtk(self, string filename, ... [, 'ascii'][, 'quality'])

Exports a mesh to a VTK file .

If 'quality' is specified, an estimation of the quality of each convex will be written to the file.

See also MeshFem.export_to_vtk(), Slice.export_to_vtk().

faces_from_cvid (*CVIDs=None*, **args*)

Synopsis: CVFIDs = Mesh.faces_from_cvid(self[, ivec CVIDs][, 'merge'])

Return a list of convexes faces from a list of convex #id.

CVFIDs is a two-rows matrix, the first row lists convex #ids, and the second lists face numbers (local number in the convex). If *CVIDs* is not given, all convexes are considered. The optional argument 'merge' merges faces shared by the convex of *CVIDs*.

faces_from_pid (*PIDs*)

Return the convex faces whose vertex #ids are in *PIDs*.

CVFIDs is a two-rows matrix, the first row lists convex #ids, and the second lists face numbers (local number in the convex). For a convex face to be returned, EACH of its points have to be listed in *PIDs*.

geotrans (*CVIDs=None*)

Returns an array of the geometric transformations.

See also Mesh.cvstruct().

max_cvid ()

Return the maximum #id of all convexes in the mesh (see 'max pid').

max_pid()
Return the maximum #id of all points in the mesh (see 'max_cvid').

memsize()
Return the amount of memory (in bytes) used by the mesh.

merge(*m2*)
Merge with the Mesh *m2*.
Overlapping points won't be duplicated. If *m2* is a MeshFem object, its linked mesh will be used.

nbcvx()
Get the number of convexes of the mesh.

nbpts()
Get the number of points of the mesh.

normal_of_face(*cv, f, nfpt=None*)
Evaluates the normal of convex *cv*, face *f* at the *nfpt* point of the face.
If *nfpt* is not specified, then the normal is evaluated at each geometrical node of the face.

normal_of_faces(*CVFIDs*)
Evaluates (at face centers) the normals of convexes.
CVFIDs is supposed a two-rows matrix, the first row lists convex #ids, and the second lists face numbers (local number in the convex).

optimize_structure()
Reset point and convex numbering.
After optimisation, the points (resp. convexes) will be consecutively numbered from 0 to `Mesh.max_pid()-1` (resp. `Mesh.max_cvid()-1`).

orphaned_pid()
Search point #id which are not linked to a convex.

outer_faces(*CVIDs=None*)
Return the faces which are not shared by two convexes.
CVFIDs is a two-rows matrix, the first row lists convex #ids, and the second lists face numbers (local number in the convex). If *CVIDs* is not given, all convexes are considered, and it basically returns the mesh boundary. If *CVIDs* is given, it returns the boundary of the convex set whose #ids are listed in *CVIDs*.

pid()
Return the list of points #id of the mesh.
Note that their numbering is not supposed to be contiguous from 0 to `Mesh.nbpts()-1`, especially if some points have been removed from the mesh. You can use `Mesh.optimize_structure()` to enforce a contiguous numbering.

pid_from_coords(*PTS, radius=0*)
Search point #id whose coordinates are listed in *PTS*.
PTS is an array containing a list of point coordinates. On return, *PIDs* is a vector containing points #id for each point found in *eps* range, and -1 for those which were not found in the mesh.

pid_from_cvid(*CVIDs=None*)
Return the points attached to each convex of the mesh.
If *CVIDs* is omitted, all the convexes will be considered (equivalent to *CVIDs* = `Mesh.max_cvid()`). *IDx* is a vector, `length(IDx) = length(CVIDs)+1`. *Pid* is a vector containing the concatenated list of #id of points of each convex in *CVIDs*. Each entry of *IDx* is the position of the corresponding convex point list in *Pid*. Hence, for example, the list of #id of points of the second convex is `Pid[IDx(2):IDx(3)]`.
If *CVIDs* contains convex #id which do not exist in the mesh, their point list will be empty.

pid_in_cvids(*CVIDs*)
Search point #id listed in *CVIDs*.
PIDs is a vector containing points #id.

pid_in_faces (*CVFIDs*)

Search point #id listed in *CVFIDs*.

CVFIDs is a two-rows matrix, the first row lists convex #ids, and the second lists face numbers. On return, *PIDs* is a vector containing points #id.

pid_in_regions (*RIDs*)

Search point #id listed in *RIDs*.

PIDs is a vector containing points #id.

pts (*PIDs=None*)

Return the list of point coordinates of the mesh.

Each column of the returned matrix contains the coordinates of one point. If the optional argument *PIDs* was given, only the points whose #id is listed in this vector are returned. Otherwise, the returned matrix will have `Mesh.max_pid()` columns, which might be greater than `Mesh.nbpts()` (if some points of the mesh have been destroyed and no call to `Mesh.optimize_structure()` have been issued). The columns corresponding to deleted points will be filled with NaN. You can use `Mesh.pid()` to filter such invalid points.

pts_from_cvid (*CVIDs=None*)

Search point listed in *CVID*.

If *CVIDs* is omitted, all the convexes will be considered (equivalent to *CVIDs* = `Mesh.max_cvid()`). *IDx* is a vector, `length(IDx) = length(CVIDs)+1`. *Pts* is a vector containing the concatenated list of points of each convex in *CVIDs*. Each entry of *IDx* is the position of the corresponding convex point list in *Pts*. Hence, for example, the list of points of the second convex is `Pts[:,IDx[2]:IDx[3]]`.

If *CVIDs* contains convex #id which do not exist in the mesh, their point list will be empty.

quality (*CVIDs=None*)

Return an estimation of the quality of each convex ($0 \leq Q \leq 1$).

refine (*CVIDs=None*)

Use a Bank strategy for mesh refinement.

If *CVIDs* is not given, the whole mesh is refined. Note that the regions, and the finite element methods and integration methods of the `MeshFem` and `MeshIm` objects linked to this mesh will be automatically refined.

region (*RIDs*)

Return the list of convexes/faces on the regions *RIDs*.

CVFIDs is a two-rows matrix, the first row lists convex #ids, and the second lists face numbers (local number in the convex). (and -1 when the whole convex is in the regions).

region_intersect (*r1, r2*)

Replace the region number *r1* with its intersection with region number *r2*.

region_merge (*r1, r2*)

Merge region number *r2* into region number *r1*.

region_substract (*r1, r2*)

Replace the region number *r1* with its difference with region number *r2*.

regions ()

Return the list of valid regions stored in the mesh.

save (*filename*)

Save the mesh object to an ascii file.

This mesh can be restored with `Mesh('load', filename)`.

set_boundary (*rnum, CVFIDs*)

DEPRECATED FUNCTION. Use 'region' instead.

set_pts (*PTS*)

Replace the coordinates of the mesh points with those given in *PTS*.

set_region (*rnum*, *CVFIDs*)

Assigns the region number *rnum* to the convex faces stored in each column of the matrix *CVFIDs*.

The first row of *CVFIDs* contains a convex #ids, and the second row contains a face number in the convex (or -1 for the whole convex (regions are usually used to store a list of convex faces, but you may also use them to store a list of convexes)).

transform (*T*)

Applies the matrix *T* to each point of the mesh.

Note that *T* is not required to be a $N \times N$ matrix (with $N = \text{Mesh.dim}()$). Hence it is possible to transform a 2D mesh into a 3D one (and reciprocally).

translate (*V*)

Translates each point of the mesh from *V*.

triangulated_surface (*Nrefine*, *CVLIST=None*)

[DEPRECATED FUNCTION! will be removed in a future release]

Similar function to `Mesh.curved_edges()` : split (if necessary, i.e. if the geometric transformation is non-linear) each face into sub-triangles and return their coordinates in T (see also `gf_compute('eval on P1 tri mesh')`)

7.11 MeshFem

class MeshFem (**args*)

GetFEM MeshFem object

This object represents a finite element method defined on a whole mesh.

General constructor for MeshFem objects

- `MF = MeshFem('load', string fname[, Mesh m])` Load a MeshFem from a file.

If the mesh *m* is not supplied (this kind of file does not store the mesh), then it is read from the file *fname* and its descriptor is returned as the second output argument.

- `MF = MeshFem('from string', string [, Mesh m])` Create a MeshFem object from its string description.

See also `MeshFem.char()`

- `MF = MeshFem('clone', MeshFem mf2)` Create a copy of a MeshFem.

- `MF = MeshFem('sum', MeshFem mf1, MeshFem mf2[, MeshFem mf3[, ...]])` Create a MeshFem that combines two (or more) MeshFem's.

All MeshFem must share the same mesh (see `Fem('interpolated_fem')` to map a MeshFem onto another). After that, you should not modify the FEM of *mf1*, *mf2* etc.

- `MF = MeshFem('levelset', MeshLevelSet mls, MeshFem mf)` Create a MeshFem that is conformal to implicit surfaces defined in MeshLevelSet.

- `MF = MeshFem('global function', Mesh m, LevelSet ls, (GlobalFunction GF1, ...)[, int Qdim_m])` Create a MeshFem whose base functions are global function given by the user.

- `MF = MeshFem('partial', MeshFem mf, ivec DOFs[, ivec RCVs])` Build a restricted MeshFem by keeping only a subset of the degrees of freedom of *mf*.

If *RCVs* is given, no FEM will be put on the convexes listed in *RCVs*.

- `MF = MeshFem(Mesh m[, int Qdim_m=1[, int Qdim_n=1]])` Build a new MeshFem object. *Qdim_m* and *Qdim_n* parameters are optional. Returns the handle of the created object.

basic_dof_from_cv (*CVids*)

Return the dof of the convexes listed in *CVids*.

WARNING: the Degree of Freedom might be returned in ANY order, do not use this function in your assembly routines. Use 'basic dof from cvid' instead, if you want to be able to map a convex number with its associated degrees of freedom.

One can also get the list of basic dof on a set on convex faces, by indicating on the second row of *CVids* the faces numbers (with respect to the convex number on the first row).

basic_dof_from_cvid (*CVids=None*)

Return the degrees of freedom attached to each convex of the mesh.

If *CVids* is omitted, all the convexes will be considered (equivalent to *CVids = 1 ... Mesh.max_cvid()*).

IDx is a vector, $length(IDx) = length(CVids)+1$. *DOFs* is a vector containing the concatenated list of dof of each convex in *CVids*. Each entry of *IDx* is the position of the corresponding convex point list in *DOFs*. Hence, for example, the list of points of the second convex is *DOFs*[*IDx*(2):*IDx*(3)].

If *CVids* contains convex #id which do not exist in the mesh, their point list will be empty.

basic_dof_nodes (*DOFids=None*)

Get location of basic degrees of freedom.

Return the list of interpolation points for the specified dof #IDs in *DOFids* (if *DOFids* is omitted, all basic dof are considered).

basic_dof_on_region (*Rs*)

Return the list of basic dof (before the optional reduction) lying on one of the mesh regions listed in *Rs*.

More precisely, this function returns the basic dof whose support is non-null on one of regions whose #ids are listed in *Rs* (note that for boundary regions, some dof nodes may not lie exactly on the boundary, for example the dof of $P_k(n,0)$ lies on the center of the convex, but the base function is not null on the convex border).

char (*opt=None*)

Output a string description of the MeshFem.

By default, it does not include the description of the linked mesh object, except if *opt* is 'with_mesh'.

convex_index ()

Return the list of convexes who have a FEM.

display ()

displays a short summary for a MeshFem object.

dof_from_cv (*CVids*)

Deprecated function. Use `MeshFem.basic_dof_from_cv()` instead.

dof_from_cvid (*CVids=None*)

Deprecated function. Use `MeshFem.basic_dof_from_cvid()` instead.

dof_from_im (*mim, p=None*)

Return a selection of dof who contribute significantly to the mass-matrix that would be computed with *mf* and the integration method *mim*.

p represents the dimension on what the integration method operates (default *p = mesh dimension*).

IMPORTANT: you still have to set a valid integration method on the convexes which are not crosses by the levelset!

dof_nodes (*DOFids=None*)

Deprecated function. Use `MeshFem.basic_dof_nodes()` instead.

dof_on_region (*Rs*)

Return the list of dof (after the optional reduction) lying on one of the mesh regions listed in *Rs*.

More precisely, this function returns the basic dof whose support is non-null on one of regions whose #ids are listed in *Rs* (note that for boundary regions, some dof nodes may not lie exactly on the boundary, for example the dof of $P_k(n,0)$ lies on the center of the convex, but the base function is not null on the convex border).

For a reduced mesh_fem a dof is lying on a region if its potential corresponding shape function is nonzero on this region. The extension matrix is used to make the correspondance between basic and reduced dofs.

dof_partition()

Get the 'dof_partition' array.

Return the array which associates an integer (the partition number) to each convex of the MeshFem. By default, it is an all-zero array. The degrees of freedom of each convex of the MeshFem are connected only to the dof of neighbouring convexes which have the same partition number, hence it is possible to create partially discontinuous MeshFem very easily.

eval (expression, gl={}, lo={})

interpolate an expression on the (lagrangian) MeshFem.

Examples:

```
mf.eval('x[0]*x[1]') # interpolates the function 'x*y'
mf.eval(' [x[0],x[1]]') # interpolates the vector field '[x,y]'
```

```
import numpy as np
```

```
mf.eval('np.sin(x[0])',globals(),locals()) # interpolates the function sin(x)
```

export_to_dx (filename, *args)

Synopsis: MeshFem.export_to_dx(self,string filename, ...['as', string mesh_name][,'edges'][,'serie',string serie_name][,'ascii'][,'append'], U, 'name'...)

Export a MeshFem and some fields to an OpenDX file.

This function will fail if the MeshFem mixes different convex types (i.e. quads and triangles), or if OpenDX does not handle a specific element type (i.e. prism connections are not known by OpenDX).

The FEM will be mapped to order 1 Pk (or Qk) FEMs. If you need to represent high-order FEMs or high-order geometric transformations, you should consider Slice.export_to_dx().

export_to_pos (filename, name=None, *args)

Synopsis: MeshFem.export_to_pos(self,string filename[, string name][[,MeshFem mf1], mat U1, string nameU1[,MeshFem mf2], mat U2, string nameU2,...])

Export a MeshFem and some fields to a pos file.

The FEM and geometric transformations will be mapped to order 1 isoparametric Pk (or Qk) FEMs (as GMSH does not handle higher order elements).

export_to_vtk (filename, *args)

Synopsis: MeshFem.export_to_vtk(self,string filename, ... ['ascii'], U, 'name'...)

Export a MeshFem and some fields to a vtk file.

The FEM and geometric transformations will be mapped to order 1 or 2 isoparametric Pk (or Qk) FEMs (as VTK does not handle higher order elements). If you need to represent high-order FEMs or high-order geometric transformations, you should consider Slice.export_to_vtk().

extension_matrix()

Return the optional extension matrix.

fem (CVids=None)

Return a list of FEM used by the MeshFem.

FEMs is an array of all Fem objects found in the convexes given in CVids. If CV2F was supplied as an output argument, it contains, for each convex listed in CVids, the index of its corresponding FEM in FEMs.

Convexes which are not part of the mesh, or convexes which do not have any FEM have their corresponding entry in CV2F set to -1.

has_linked_mesh_levelset()

Is a mesh_fem_level_set or not.

interpolate_convex_data (Ucv)

Interpolate data given on each convex of the mesh to the MeshFem dof. The MeshFem has to be lagrangian, and should be discontinuous (typically a FEM_PK(N,0) or FEM_QK(N,0) should be used).

The last dimension of the input vector `Ucv` should have `Mesh.max_cvid()` elements.

Example of use: `MeshFem.interpolate_convex_data(Mesh.quality())`

is_equivalent (*CVids=None*)

Test if the MeshFem is equivalent.

See `MeshFem.is_lagrangian()`

is_lagrangian (*CVids=None*)

Test if the MeshFem is Lagrangian.

Lagrangian means that each base function $\Phi[i]$ is such that $\Phi[i](P[j]) = \delta(i,j)$, where $P[j]$ is the dof location of the j th base function, and $\delta(i,j) = 1$ if $i=j$, else 0.<Par>

If *CVids* is omitted, it returns 1 if all convexes in the mesh are Lagrangian. If *CVids* is used, it returns the convex indices (with respect to *CVids*) which are Lagrangian.

is_polynomial (*CVids=None*)

Test if all base functions are polynomials.

See `MeshFem.is_lagrangian()`

is_reduced ()

Return 1 if the optional reduction matrix is applied to the dofs.

linked_mesh ()

Return a reference to the Mesh object linked to *mf*.

linked_mesh_levelset ()

if it is a `mesh_fem_level_set` gives the linked `mesh_level_set`.

memsize ()

Return the amount of memory (in bytes) used by the `mesh_fem` object.

The result does not take into account the linked mesh object.

mesh ()

Return a reference to the Mesh object linked to *mf*. (identical to `Mesh.linked_mesh()`)

nb_basic_dof ()

Return the number of basic degrees of freedom (dof) of the MeshFem.

nbdof ()

Return the number of degrees of freedom (dof) of the MeshFem.

non_conformal_basic_dof (*CVids=None*)

Return partially linked degrees of freedom.

Return the basic dof located on the border of a convex and which belong to only one convex, except the ones which are located on the border of the mesh. For example, if the convex 'a' and 'b' share a common face, 'a' has a P1 FEM, and 'b' has a P2 FEM, then the basic dof on the middle of the face will be returned by this function (this can be useful when searching the interfaces between classical FEM and hierarchical FEM).

non_conformal_dof (*CVids=None*)

Deprecated function. Use `MeshFem.non_conformal_basic_dof()` instead.

qdim ()

Return the dimension *Q* of the field interpolated by the MeshFem.

By default, *Q*=1 (scalar field). This has an impact on the dof numbering.

reduction (*s*)

Set or unset the use of the reduction/extension matrices.

reduction_matrices (*R, E*)

Set the reduction and extension matrices and valid their use.

reduction_matrix ()

Return the optional reduction matrix.

save (*filename*, *opt=None*)

Save a MeshFem in a text file (and optionally its linked mesh object if *opt* is the string 'with_mesh').

set_classical_discontinuous_fem (*K*, *alpha=None*, **args*)

Synopsis: MeshFem.set_classical_discontinuous_fem(self, int K[, @tscalar alpha[, ivec CVIDX]])

Assigns a classical (Lagrange polynomial) discontinuous fem of order K.

Similar to MeshFem.classical_fem() except that FEM_PK_DISCONTINUOUS is used. Param *alpha* the node inset, $0 \leq \alpha < 1$, where 0 implies usual dof nodes, greater values move the nodes toward the center of gravity, and 1 means that all degrees of freedom collapse on the center of gravity.

set_classical_fem (*k*, *CVids=None*)

Assign a classical (Lagrange polynomial) fem of order *k* to the MeshFem.

Uses FEM_PK for simplexes, FEM_QK for parallelepipeds etc.

set_dof_partition (*DOFP*)

Change the 'dof_partition' array.

DOFP is a vector holding a integer value for each convex of the MeshFem. See MeshFem.dof_partition() for a description of "dof partition".

set_fem (*f*, *CVids=None*)

Set the Finite Element Method.

Assign a FEM *f* to all convexes whose #ids are listed in *CVids*. If *CVids* is not given, the integration is assigned to all convexes.

See the help of Fem to obtain a list of available FEM methods.

set_partial (*DOFs*, *RCVs=None*)

Can only be applied to a partial MeshFem. Change the subset of the degrees of freedom of *mf*.

If *RCVs* is given, no FEM will be put on the convexes listed in *RCVs*.

set_qdim (*Q*)

Change the *Q* dimension of the field that is interpolated by the MeshFem.

Q = 1 means that the MeshFem describes a scalar field, *Q = N* means that the MeshFem describes a vector field of dimension N.

7.12 MeshIm

class MeshIm (**args*)

GetFEM MeshIm object

This object represent an integration method defined on a whole mesh (an potentially on its boundaries).

General constructor for MeshIm objects

- MeshIm('load', string fname[, Mesh m]) Load a MeshIm from a file.

If the mesh *m* is not supplied (this kind of file does not store the mesh), then it is read from the file and its descriptor is returned as the second output argument.

- MeshIm('from string', string s[, mesh M]) Create a MeshIm object from its string description.

See also MeshIm.char()

- MeshIm('clone', MeshIm mim2) Create a copy of a MeshIm.

- MeshIm('levelset', MeshLevelSet mls, string where, Integ im[, Integ im_tip[, Integ im_set]]) Build an integration method conformal to a partition defined implicitly by a levelset.

The *where* argument define the domain of integration with respect to the levelset, it has to be chosen among 'ALL', 'INSIDE', 'OUTSIDE' and 'BOUNDARY'.

•**MeshIm**(Mesh *m*, [{Integ *im*|int *im_degree*}]) Build a new MeshIm object.

For convenience, optional arguments (*im* or *im_degree*) can be provided, in that case a call to **MeshIm.integ()** is issued with these arguments.

adapt ()

For a MeshIm levelset object only. Adapt the integration methods to a change of the levelset function.

char ()

Output a string description of the MeshIm.

By default, it does not include the description of the linked Mesh object.

convex_index ()

Return the list of convexes who have a integration method.

Convexes who have the dummy IM_NONE method are not listed.

display ()

displays a short summary for a MeshIm object.

eltm (*em*, *cv*, *f*=None)

Return the elementary matrix (or tensor) integrated on the convex *cv*.

WARNING

Be sure that the fem used for the construction of *em* is compatible with the fem assigned to element *cv* ! This is not checked by the function ! If the argument *f* is given, then the elementary tensor is integrated on the face *f* of *cv* instead of the whole convex.

im_nodes (*CVids*=None)

Return the coordinates of the integration points, with their weights.

CVids may be a list of convexes, or a list of convex faces, such as returned by **Mesh.region()**

WARNING

Convexes which are not part of the mesh, or convexes which do not have an approximate integration method don't have their corresponding entry (this has no meaning for exact integration methods!).

integ (*CVids*=None)

Return a list of integration methods used by the MeshIm.

I is an array of all Integ objects found in the convexes given in *CVids*. If *CV2I* was supplied as an output argument, it contains, for each convex listed in *CVids*, the index of its corresponding integration method in *I*.

Convexes which are not part of the mesh, or convexes which do not have any integration method have their corresponding entry in *CV2I* set to -1.

linked_mesh ()

Returns a reference to the Mesh object linked to *mim*.

memsize ()

Return the amount of memory (in bytes) used by the MeshIm object.

The result does not take into account the linked Mesh object.

save (*filename*)

Saves a MeshIm in a text file (and optionally its linked mesh object).

set_integ (*args)

Synopsis: **MeshIm.set_integ**(self,{Integ *im*|int *im_degree*}[, ivec *CVids*])

Set the integration method.

Assign an integration method to all convexes whose #ids are listed in *CVids*. If *CVids* is not given, the integration is assigned to all convexes. It is possible to assign a specific integration method with an integration method handle *im* obtained via **Integ**('IM_SOMETHING'), or to let getfem choose a suitable integration method with *im_degree* (chosen such that polynomials of *degree* ≤ *im_degree* are exactly integrated. If *im_degree*=-1, then the dummy integration method IM_NONE will be used.)

7.13 MeshLevelSet

class MeshLevelSet (*args)

GetFEM MeshLevelSet object

General constructor for mesh_levelset objects. The role of this object is to provide a mesh cut by a certain number of level_set. This object is used to build conformal integration method (object mim and enriched finite element methods (Xfem)).

General constructor for MeshLevelSet objects

•**MLS** = MeshLevelSet (Mesh m) Build a new MeshLevelSet object from a Mesh and returns its handle.

adapt ()

Do all the work (cut the convexes with the levelsets).

To initialize the MeshLevelSet object or to actualize it when the value of any levelset function is modified, one has to call this method.

add (ls)

Add a link to the LevelSet ls.

Only a reference is kept, no copy is done. In order to indicate that the linked Mesh is cut by a LevelSet one has to call this method, where ls is an LevelSet object. An arbitrary number of LevelSet can be added.

WARNING

The Mesh of ls and the linked Mesh must be the same.

char ()

Output a (unique) string representation of the MeshLevelSetn.

This can be used to perform comparisons between two different MeshLevelSet objects. This function is to be completed.

crack_tip_convexes ()

Return the list of convex #id's of the linked Mesh on which have a tip of any linked LevelSet's.

cut_mesh ()

Return a Mesh cut by the linked LevelSet's.

display ()

displays a short summary for a MeshLevelSet object.

levelsets ()

Return a list of references to the linked LevelSet's.

linked_mesh ()

Return a reference to the linked Mesh.

memsize ()

Return the amount of memory (in bytes) used by the MeshLevelSet.

nb_ls ()

Return the number of linked LevelSet's.

sup (ls)

Remove a link to the LevelSet ls.

7.14 Model

class Model (*args)

GetFEM Model object

Model variables store the variables and the state data and the description of a model. This includes the global tangent matrix, the right hand side and the constraints. There are two kinds of models, the *real* and the *complex* models.

Model object is the evolution for Getfem++ 4.0 of the MdState object.

General constructor for Model objects

- MDS = Model('real') Build a model for real unknowns.
- MDS = Model('complex') Build a model for complex unknowns.

add_Dirichlet_condition_with_multipliers (*mim*, *varname*, *mult_description*, *region*, *dataname=None*)

Add a Dirichlet condition on the variable *varname* and the mesh region *region*. This region should be a boundary. The Dirichlet condition is prescribed with a multiplier variable described by *mult_description*. If *mult_description* is a string this is assumed to be the variable name corresponding to the multiplier (which should be first declared as a multiplier variable on the mesh region in the model). If it is a finite element method (mesh_fem object) then a multiplier variable will be added to the model and build on this finite element method (it will be restricted to the mesh region *region* and eventually some conflicting dofs with some other multiplier variables will be suppressed). If it is an integer, then a multiplier variable will be added to the model and build on a classical finite element of degree that integer. *dataname* is the optional right hand side of the Dirichlet condition. It could be constant or described on a fem; scalar or vector valued, depending on the variable on which the Dirichlet condition is prescribed. Return the brick index in the model.

add_Dirichlet_condition_with_penalization (*mim*, *varname*, *coeff*, *region*, *dataname=None*)

Add a Dirichlet condition on the variable *varname* and the mesh region *region*. This region should be a boundary. The Dirichlet condition is prescribed with penalization. The penalization coefficient is initially *coeff* and will be added to the data of the model. *dataname* is the optional right hand side of the Dirichlet condition. It could be constant or described on a fem; scalar or vector valued, depending on the variable on which the Dirichlet condition is prescribed. Return the brick index in the model.

add_Fourier_Robin_brick (*mim*, *varname*, *dataname*, *region*)

Add a Fourier-Robin term to the model relatively to the variable *varname*. This corresponds to a weak term of the form $\int (qu).v$. *dataname* should contain the parameter *q* of the Fourier-Robin condition. *region* is the mesh region on which the term is added. Return the brick index in the model.

add_Helmholtz_brick (*mim*, *varname*, *dataname*, *region=None*)

Add a Helmholtz term to the model relatively to the variable *varname*. *dataname* should contain the wave number. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_Laplacian_brick (*mim*, *varname*, *region=None*)

Add a Laplacian term to the model relatively to the variable *varname*. If this is a vector valued variable, the Laplacian term is added componentwise. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_basic_contact_brick (*varname_u*, *multname_n*, *multname_t=None*, *args)

Synopsis: ind = Model.add_basic_contact_brick(self, string varname_u, string multname_n[, string multname_t], string dataname_r, Spmat BN[, Spmat BT, string dataname_friction_coeff][, string dataname_gap[, string dataname_alpha[, int symmetrized]])

Add a contact with or without friction brick to the model. If *U* is the vector of degrees of freedom on which the unilateral constraint is applied, the matrix *BN* have to be such that this constraint is defined by $B_N U \leq 0$. A friction condition can be considered by adding the three parameters *multname_t*, *BT* and *dataname_friction_coeff*. In this case, the tangential displacement is $B_T U$ and the matrix *BT* should have as many rows as *BN* multiplied by $d - 1$ where d is the domain dimension. In this case also, *dataname_friction_coeff* is a data which represents the coefficient of friction. It can be a scalar or a vector representing a value on each contact condition. The unilateral constraint is prescribed thank to a multiplier *multname_n* whose dimension should be equal to the number of rows of *BN*. If a friction condition

is added, it is prescribed with a multiplier *multname_t* whose dimension should be equal to the number of rows of *BT*. The augmentation parameter *r* should be chosen in a range of acceptable values (see Getfem user documentation). *dataname_gap* is an optional parameter representing the initial gap. It can be a single value or a vector of value. *dataname_alpha* is an optional homogenization parameter for the augmentation parameter (see Getfem user documentation). The parameter *symmetrized* indicates that the symmetry of the tangent matrix will be kept or not (except for the part representing the coupling between contact and friction which cannot be symmetrized).

add_basic_d2_on_dt2_brick (*mim*, *varnameU*, *datanameV*, *dataname_dt*, *dataname_alpha*,
dataname_rho=None, *args)

Synopsis: `ind = Model.add_basic_d2_on_dt2_brick(self, MeshIm mim, string varnameU, string datanameV, string dataname_dt, string dataname_alpha[, string dataname_rho[, int region]])`

Add the standard discretization of a second order time derivative on *varnameU*. *datanameV* is a data represented on the same finite element method as *U* which represents the time derivative of *U*. The parameter *dataname_rho* is the density which could be omitted (the default value is 1). This brick should be used in addition to a time dispatcher for the other terms. The time derivative *v* of the variable *u* is preferably computed as a post-treatment which depends on each scheme. The parameter *dataname_alpha* depends on the time integration scheme. Return the brick index in the model.

add_basic_d_on_dt_brick (*mim*, *varnameU*, *dataname_dt*, *dataname_rho*=None, *args)

Synopsis: `ind = Model.add_basic_d_on_dt_brick(self, MeshIm mim, string varnameU, string dataname_dt[, string dataname_rho[, int region]])`

Add the standard discretization of a first order time derivative on *varnameU*. The parameter *dataname_rho* is the density which could be omitted (the default value is 1). This brick should be used in addition to a time dispatcher for the other terms. Return the brick index in the model.

add_constraint_with_multipliers (*varname*, *multname*, *B*, *L*)

Add an additional explicit constraint on the variable *varname* thanks to a multiplier *multname* previously added to the model (should be a fixed size variable). The constraint is $BU = L$ with *B* being a rectangular sparse matrix. It is possible to change the constraint at any time with the methods `Model.set_private_matrix()` and `Model.set_private_rhs()`. Return the brick index in the model.

add_constraint_with_penalization (*varname*, *coeff*, *B*, *L*)

Add an additional explicit penalized constraint on the variable *varname*. The constraint is $BU=L$ with *B* being a rectangular sparse matrix. Be aware that *B* should not contain a plain row, otherwise the whole tangent matrix will be plain. It is possible to change the constraint at any time with the methods `Model.set_private_matrix()` and `Model.set_private_rhs()`. The method `Model.change_penalization_coeff()` can be used. Return the brick index in the model.

add_contact_with_rigid_obstacle_brick (*mim*, *varname_u*, *multname_n*, *multname_t*=None, *args)

Synopsis: `ind = Model.add_contact_with_rigid_obstacle_brick(self, MeshIm mim, string varname_u, string multname_n[, string multname_t], string dataname_r[, string dataname_friction_coeff], int region, string obstacle[, int symmetrized])`

Add a contact with or without friction condition with a rigid obstacle to the model. The condition is applied on the variable *varname_u* on the boundary corresponding to *region*. The rigid obstacle should be described with the string *obstacle* being a signed distance to the obstacle. This string should be an expression where the coordinates are 'x', 'y' in 2D and 'x', 'y', 'z' in 3D. For instance, if the rigid obstacle correspond to $z \leq 0$, the corresponding signed distance will be simply "z". *multname_n* should be a fixed size variable whose size is the number of degrees of freedom on boundary *region*. It represent the contact equivalent nodal forces. In order to add a friction condition one has to add the *multname_t* and *dataname_friction_coeff* parameters. *multname_t* should be a fixed size variable whose size is the number of degrees of freedom on boundary *region* multiplied by $d - 1$ where *d* is the domain dimension. It represent the friction equivalent nodal forces. The augmentation parameter *r* should be chosen in a range of acceptable values (close to the Young modulus of the elastic body, see Getfem user documentation). *dataname_friction_coeff* is the friction coefficient. It could be a scalar or a vector of values representing the friction coefficient on each contact node. The parameter *symmetrized* indicates that the symmetry of the tangent matrix will be kept or not. Basically, this brick compute the matrix *BN* and the vectors *gap* and *alpha* and calls the basic contact brick.

add_data (*name, size, niter=None*)

Add a data to the model of constant size. *name* is the data name and *niter* is the optional number of version of the data stored, for time integration schemes.

add_explicit_matrix (*varname1, varname2, B, issymmetric=None, *args*)

Synopsis: `ind = Model.add_explicit_matrix(self, string varname1, string varname2, Spmat B[, int issymmetric[, int iscoercive]])`

Add a brick representing an explicit matrix to be added to the tangent linear system relatively to the variables '*varname1*' and '*varname2*'. The given matrix should have as many rows as the dimension of '*varname1*' and as many columns as the dimension of '*varname2*'. If the two variables are different and if *issymmetric* is set to 1 then the transpose of the matrix is also added to the tangent system (default is 0). Set *iscoercive* to 1 if the term does not affect the coercivity of the tangent system (default is 0). The matrix can be changed by the command `Model.set_private_matrix()`. Return the brick index in the model.

add_explicit_rhs (*varname, L*)

Add a brick representing an explicit right hand side to be added to the right hand side of the tangent linear system relatively to the variable '*varname*'. The given rhs should have the same size than the dimension of '*varname*'. The rhs can be changed by the command `Model.set_private_rhs()`. Return the brick index in the model.

add_fem_data (*name, mf, qdim=None, *args*)

Synopsis: `Model.add_fem_data(self, string name, MeshFem mf[, int qdim[, int niter]])`

Add a data to the model linked to a MeshFem. *name* is the data name, *qdim* is the optional dimension of the data over the MeshFem and *niter* is the optional number of version of the data stored, for time integration schemes.

add_fem_variable (*name, mf, niter=None*)

Add a variable to the model linked to a MeshFem. *name* is the variable name and *niter* is the optional number of version of the data stored, for time integration schemes.

add_generalized_Dirichlet_condition_with_multipliers (*mim, varname, mult_description, region, dataname, Hname*)

Add a Dirichlet condition on the variable *varname* and the mesh region *region*. This version is for vector field. It prescribes a condition $Hu = r$ where H is a matrix field. The region should be a boundary. The Dirichlet condition is prescribed with a multiplier variable described by *mult_description*. If *mult_description* is a string this is assumed to be the variable name corresponding to the multiplier (which should be first declared as a multiplier variable on the mesh region in the model). If it is a finite element method (mesh_fem object) then a multiplier variable will be added to the model and build on this finite element method (it will be restricted to the mesh region *region* and eventually some conflicting dofs with some other multiplier variables will be suppressed). If it is an integer, then a multiplier variable will be added to the model and build on a classical finite element of degree that integer. *dataname* is the right hand side of the Dirichlet condition. It could be constant or described on a fem; scalar or vector valued, depending on the variable on which the Dirichlet condition is prescribed. *Hname* is the data corresponding to the matrix field ' H '. Return the brick index in the model.

add_generalized_Dirichlet_condition_with_penalization (*mim, varname, coeff, region, dataname, Hname*)

Add a Dirichlet condition on the variable *varname* and the mesh region *region*. This version is for vector field. It prescribes a condition $Hu = r$ where H is a matrix field. The region should be a boundary. The Dirichlet condition is prescribed with penalization. The penalization coefficient is initially *coeff* and will be added to the data of the model. *dataname* is the right hand side of the Dirichlet condition. It could be constant or described on a fem; scalar or vector valued, depending on the variable on which the Dirichlet condition is prescribed. *Hname* is the data corresponding to the matrix field ' H '. It has to be a constant matrix or described on a scalar fem. Return the brick index in the model.

add_generic_elliptic_brick (*mim, varname, dataname, region=None*)

Add a generic elliptic term to the model relatively to the variable *varname*. The shape of the elliptic term depends both on the variable and the data. This corresponds to a term $-\text{div}(a \nabla u)$ where a is the data and u the variable. The data can be a scalar, a matrix or an order four tensor. The variable can be vector

valued or not. If the data is a scalar or a matrix and the variable is vector valued then the term is added componentwise. An order four tensor data is allowed for vector valued variable only. The data can be constant or described on a fem. Of course, when the data is a tensor describe on a finite element method (a tensor field) the data can be a huge vector. The components of the matrix/tensor have to be stored with the fortran order (columnwise) in the data vector (compatibility with blas). The symmetry of the given matrix/tensor is not verified (but assumed). If this is a vector valued variable, the Laplacian term is added componentwise. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_initialized_data (*name*, *V*)

Add a fixed size data to the model linked to a MeshFem. *name* is the data name and *V* is the value of the data.

add_initialized_fem_data (*name*, *mf*, *V*)

Add a data to the model linked to a MeshFem. *name* is the data name. The data is initialized with *V*. The data can be a scalar or vector field.

add_isotropic_linearized_elasticity_brick (*mim*, *varname*, *dataname_lambda*, *dataname_mu*, *region=None*)

Add an isotropic linearized elasticity term to the model relatively to the variable *varname*. *dataname_lambda* and *dataname_mu* should contain the Lam'e coefficients. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_linear_incompressibility_brick (*mim*, *varname*, *multname_pressure*, *region=None*, **args*)

Synopsis: `ind = Model.add_linear_incompressibility_brick(self, MeshIm mim, string varname, string multname_pressure[, int region[, string dataname_coeff]])`

Add a linear incompressibility condition on *variable*. *multname_pressure* is a variable which represent the pressure. Be aware that an inf-sup condition between the finite element method describing the pressure and the primal variable has to be satisfied. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. *dataname_coeff* is an optional penalization coefficient for nearly incompressible elasticity for instance. In this case, it is the inverse of the Lam'e coefficient λ . Return the brick index in the model.

add_mass_brick (*mim*, *varname*, *dataname_rho=None*, **args*)

Synopsis: `ind = Model.add_mass_brick(self, MeshIm mim, string varname[, string dataname_rho[, int region]])`

Add mass term to the model relatively to the variable *varname*. If specified, the data *dataname_rho* should contain the density (1 if omitted). *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_midpoint_dispatcher (*bricks_indices*)

Add a midpoint time dispatcher to a list of bricks. For instance, a nonlinear term $K(U)$ will be replaced by $K((U^{n+1} + U^n)/2)$.

add_multiplier (*name*, *mf*, *primalname*, *niter=None*)

Add a particular variable linked to a fem being a multiplier with respect to a primal variable. The dof will be filtered with the `gmm::range_basis` function applied on the terms of the model which link the multiplier and the primal variable. This in order to retain only linearly independant constraints on the primal variable. Optimized for boundary multipliers. *niter* is the optional number of version of the data stored, for time integration schemes.

add_nonlinear_elasticity_brick (*mim*, *varname*, *constitutive_law*, *dataname*, *region=None*)

Add a nonlinear elasticity term to the model relatively to the variable *varname*. *lawname* is the constitutive law which could be 'Saint Venant Kirchhoff', 'Mooney Rivlin' or 'Ciarlet Geymonat'. *dataname* is a vector of parameters for the constitutive law. Its length depends on the law. It could be a short vector of constant values or a vector field described on a finite element method for variable coefficients. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_nonlinear_incompressibility_brick (*mim*, *varname*, *multname_pressure*, *region=None*)

Add an nonlinear incompressibility condition on *variable* (for large strain elasticity). *multname_pressure* is a variable which represent the pressure. Be aware that an inf-sup condition between the finite element method describing the pressure and the primal variable has to be satisfied. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_normal_source_term_brick (*mim*, *varname*, *dataname*, *region*)

Add a source term on the variable *varname* on a boundary *region*. This region should be a boundary. The source term is represented by the data *dataname* which could be constant or described on a fem. A scalar product with the outward normal unit vector to the boundary is performed. The main aim of this brick is to represent a Neumann condition with a vector data without performing the scalar product with the normal as a pre-processing. Return the brick index in the model.

add_source_term_brick (*mim*, *varname*, *dataname*, *region=None*, **args*)

Synopsis: `ind = Model.add_source_term_brick(self, MeshIm mim, string varname, string dataname[, int region[, string directdataname]])`

Add a source term to the model relatively to the variable *varname*. The source term is represented by the data *dataname* which could be constant or described on a fem. *region* is an optional mesh region on which the term is added. An additional optional data *directdataname* can be provided. The corresponding data vector will be directly added to the right hand side without assembly. Return the brick index in the model.

add_theta_method_dispatcher (*bricks_indices*, *theta*)

Add a theta-method time dispatcher to a list of bricks. For instance, a matrix term K will be replaced by $\theta KU^{n+1} + (1 - \theta)KU^n$.

add_unilateral_contact_brick (*mim1*, *mim2=None*, **args*)

Synopsis: `ind = Model.add_unilateral_contact_brick(self, MeshIm mim1[, MeshIm mim2], string varname_u1[, string varname_u2], string multname_n[, string multname_t], string dataname_r[, string dataname_fr], int rg1, int rg2[, int slave1, int slave2, int symmetrized])`

Add a contact with or without friction condition between two faces of one or two elastic bodies. The condition is applied on the variable *varname_u1* or the variables *varname_u1* and *varname_u2* depending if a single or two distinct displacement fields are given. Integers *rg1* and *rg2* represent the regions expected to come in contact with each other. In the single displacement variable case the regions defined in both *rg1* and *rg2* refer to the variable *varname_u1*. In the case of two displacement variables, *rg1* refers to *varname_u1* and *rg2* refers to *varname_u2*. *multname_n* should be a fixed size variable whose size is the number of degrees of freedom on those regions among the ones defined in *rg1* and *rg2* which are characterized as “slaves”. It represents the contact equivalent nodal normal forces. *multname_t* should be a fixed size variable whose size corresponds to the size of *multname_n* multiplied by `qdim - 1`. It represents the contact equivalent nodal tangent (frictional) forces. The augmentation parameter *r* should be chosen in a range of acceptable values (close to the Young modulus of the elastic body, see Getfem user documentation). The friction coefficient stored in the parameter *fr* is either a single value or a vector of the same size as *multname_n*. The optional parameters *slave1* and *slave2* declare if the regions defined in *rg1* and *rg2* are correspondingly considered as “slaves”. By default *slave1* is true and *slave2* is false, i.e. *rg1* contains the slave surfaces, while ‘rg2’ the master surfaces. Preferably only one of *slave1* and *slave2* is set to true. The parameter *symmetrized* indicates that the symmetry of the tangent matrix will be kept or not. Basically, this brick computes the matrices BN and BT and the vectors gap and alpha and calls the basic contact brick.

add_variable (*name*, *size*, *niter=None*)

Add a variable to the model of constant size. *name* is the variable name and *niter* is the optional number of version of the data stored, for time integration schemes.

assembly (*option=None*)

Assembly of the tangent system taking into account the terms from all bricks. *option*, if specified, should be ‘build_all’, ‘build_rhs’ or ‘build_matrix’. The default is to build the whole tangent linear system (matrix and rhs). This function is usefull to solve your problem with you own solver.

change_penalization_coeff (*ind_brick*, *coeff*)

Change the penalization coefficient of a Dirichlet condition with penalization brick. If the brick is not of this kind, this function has an undefined behavior.

char ()

Output a (unique) string representation of the Model.

This can be used to perform comparisons between two different Model objects. This function is to be completed.

clear ()

Clear the model.

compute_Von_Mises_or_Tresca (*varname*, *lawname*, *dataname*, *mf_vm*, *version=None*)

Compute on *mf_vm* the Von-Mises stress or the Tresca stress of a field for nonlinear elasticity in 3D. *lawname* is the constitutive law which could be 'SaintVenant Kirchhoff', 'Mooney Rivlin' or 'Ciarlet Geymonat'. *dataname* is a vector of parameters for the constitutive law. Its length depends on the law. It could be a short vector of constant values or a vector field described on a finite element method for variable coefficients. *version* should be 'Von_Mises' or 'Tresca' ('Von_Mises' is the default).

compute_isotropic_linearized_Von_Mises_or_Tresca (*varname*, *dataname_lambda*, *dataname_mu*, *mf_vm*, *version=None*)

Compute the Von-Mises stress or the Tresca stress of a field (only valid for isotropic linearized elasticity in 3D). *version* should be 'Von_Mises' or 'Tresca' ('Von_Mises' is the default).

contact_brick_set_BN (*indbrick*, *BN*)

Can be used to set the BN matrix of a basic contact/friction brick.

contact_brick_set_BT (*indbrick*, *BT*)

Can be used to set the BT matrix of a basic contact with friction brick.

disable_bricks (*bricks_indices*)

Disable a brick (the brick will no longer participate to the building of the tangent linear system).

display ()

displays a short summary for a Model object.

first_iter ()

To be executed before the first iteration of a time integration scheme.

from_variables ()

Return the vector of all the degrees of freedom of the model consisting of the concatenation of the variables of the model (usefull to solve your problem with you own solver).

interval_of_variable (*varname*)

Gives the interval of the variable *varname* in the linear system of the model.

is_complex ()

Return 0 if the model is real, 1 if it is complex.

listbricks ()

print to the output the list of bricks of the model.

listvar ()

print to the output the list of variables and constants of the model.

matrix_term (*ind_brick*, *ind_term*)

Gives the matrix term *ind_term* of the brick *ind_brick* if it exists

memsize ()

Return a rough approximation of the amount of memory (in bytes) used by the model.

mult_varname_Dirichlet (*ind_brick*)

Gives the name of the multiplier variable for a Dirichlet brick. If the brick is not a Dirichlet condition with multiplier brick, this function has an undefined behavior

next_iter ()

To be executed at the end of each iteration of a time integration scheme.

resize_variable (*name*, *size*)
 Resize a constant size variable of the model. *name* is the variable name.

rhs ()
 Return the right hand side of the tangent problem.

set_private_matrix (*indbrick*, *B*)
 For some specific bricks having an internal sparse matrix (explicit bricks: ‘constraint brick’ and ‘explicit matrix brick’), set this matrix.

set_private_rhs (*indbrick*, *B*)
 For some specific bricks having an internal right hand side vector (explicit bricks: ‘constraint brick’ and ‘explicit rhs brick’), set this rhs.

set_variable (*name*, *V*, *niter=None*)
 Set the value of a variable or data. *name* is the data name and *niter* is the optional number of version of the data stored, for time integration schemes.

solve (**args*)
 Synopsis: Model.solve(self[, ...])
 Run the standard getfem solver.
 Note that you should be able to use your own solver if you want (it is possible to obtain the tangent matrix and its right hand side with the Model.tangent_matrix() etc.).
 Various options can be specified:

- ‘noisy’ or ‘very_noisy’ the solver will display some information showing the progress (residual values etc.).
- ‘max_iter’, int **NIT** set the maximum iterations numbers.
- ‘max_res’, @float **RES** set the target residual value.
- ‘lsolver’, string **SOLVER_NAME** select explicitly the solver used for the linear systems (the default value is ‘auto’, which lets getfem choose itself). Possible values are ‘superlu’, ‘mumps’ (if supported), ‘cg/ildlt’, ‘gmres/ilu’ and ‘gmres/ilut’.

tangent_matrix ()
 Return the tangent matrix stored in the model .

to_variables (*V*)
 Set the value of the variables of the model with the vector *V*. Typically, the vector *V* results of the solve of the tangent linear system (usefull to solve your problem with you own solver).

unable_bricks (*bricks_indices*)
 Unable a disabled brick.

variable (*name*, *niter=None*)
 Gives the value of a variable or data.

velocity_update_for_Newmark_scheme (*id2dt2_brick*, *varnameU*, *datanameV*, *dataname_dt*, *dataname_twobeta*, *dataname_alpha*)
 Function which update the velocity v^{n+1} after the computation of the displacement u^{n+1} and before the next iteration. Specific for Newmark scheme and when the velocity is included in the data of the model.*
 This version inverts the mass matrix by a conjugate gradient.

velocity_update_for_order_two_theta_method (*varnameU*, *datanameV*, *dataname_dt*, *dataname_theta*)
 Function which update the velocity v^{n+1} after the computation of the displacement u^{n+1} and before the next iteration. Specific for theta-method and when the velocity is included in the data of the model.

7.15 Precond

class Precond (**args*)
 GetFEM Precond object

The preconditioners may store REAL or COMPLEX values. They accept getfem sparse matrices and Matlab sparse matrices.

General constructor for Precond objects

- `Precond('identity')` Create a REAL identity preconditioner.
- `Precond('cidentity')` Create a COMPLEX identity preconditioner.
- `Precond('diagonal', vec D)` Create a diagonal preconditioner.
- `Precond('ildlt', SpMat m)` Create an ILDLT (Cholesky) preconditioner for the (symmetric) sparse matrix *m*. This preconditioner has the same sparsity pattern than *m* (no fill-in).
- `Precond('ilu', SpMat m)` Create an ILU (Incomplete LU) preconditioner for the sparse matrix *m*. This preconditioner has the same sparsity pattern than *m* (no fill-in).
- `Precond('ildlth', SpMat m[, int fillin[, scalar threshold]])` Create an ILDLT (Cholesky with filling) preconditioner for the (symmetric) sparse matrix *m*. The preconditioner may add at most *fillin* additional non-zero entries on each line. The default value for *fillin* is 10, and the default threshold is $1e-7$.
- `Precond('iluth', SpMat m[, int fillin[, scalar threshold]])` Create an ILUT (Incomplete LU with filling) preconditioner for the sparse matrix *m*. The preconditioner may add at most *fillin* additional non-zero entries on each line. The default value for *fillin* is 10, and the default threshold is $1e-7$.
- `Precond('superlu', SpMat m)` Uses SuperLU to build an exact factorization of the sparse matrix *m*. This preconditioner is only available if the getfem-interface was built with SuperLU support. Note that LU factorization is likely to eat all your memory for 3D problems.
- `Precond('spmat', SpMat M)` Preconditionner given explicitly by a sparse matrix.

char()

Output a (unique) string representation of the Precond.

This can be used to perform comparisons between two different Precond objects. This function is to be completed.

display()

displays a short summary for a Precond object.

is_complex()

Return 1 if the preconditioner stores complex values.

mult(V)

Apply the preconditioner to the supplied vector.

size()

Return the dimensions of the preconditioner.

tmult(V)

Apply the transposed preconditioner to the supplied vector.

type()

Return a string describing the type of the preconditioner ('ilu', 'ildlt',...).

7.16 Slice

class Slice(*args)

GetFEM Slice object

Creation of a mesh slice. Mesh slices are very similar to a P1-discontinuous mesh_fem on which interpolation is very fast. The slice is built from a mesh object, and a description of the slicing operation, for example,

`sl = Slice(('planar',+1,[[0],[0]],[[0],[1]]), m, 5)`

cuts the original mesh with the half space $\{y>0\}$. Each convex of the original mesh m is simplexified (for example a quadrangle is splitted into 2 triangles), and each simplex is refined 5 times.

Slicing operations can be:

- cutting with a plane, a sphere or a cylinder

- intersection or union of slices
- isovalues surfaces/volumes
- “points”, “streamlines” (see below)

If the first argument is a MeshFem mf instead of a Mesh, and if it is followed by a mf -field U , then the deformation U will be applied to the mesh before the slicing operation.

The first argument can also be a slice.

Slicing operations: Always specify them between braces (i.e. in a cell array). The first argument is the name of the operation, followed the slicing options.

•{ ‘none’ }

Does not cut the mesh.

•{ ‘planar’, orient, p, n }

Planar cut. p and n define a half-space, p being a point belong to the boundary of the half-space, and n being its normal. If orient is equal to -1 (resp. 0, +1), then the slicing operation will cut the mesh with the “interior” (resp. “boundary”, “exterior”) of the half-space. Orient may also be set to +2 which means that the mesh will be sliced, but both the outer and inner parts will be kept.

•{ ‘ball’, orient, c, r }

Cut with a ball of center c and radius r .

•{ ‘cylinder’, orient, p1, p2, r }

Cut with a cylinder whose axis is the line $(p1,p2)$ and whose radius is r .

•{ ‘isovalues’,orient, mesh_fem MF, vec U, scalar V }

Cut using the isosurface of the field U (defined on the mesh_fem MF). The result is the set $\{x \text{ such that } U(x) \leq V\}$ or $\{x \text{ such that } U(x) == V\}$ or $\{x \text{ such that } U(x) < V\}$ depending on the value of ORIENT.

•{ ‘boundary’[, SLICEOP] }

Return the boundary of the result of SLICEOP, where SLICEOP is any slicing operation. If SLICEOP is not specified, then the whole mesh is considered (i.e. it is equivalent to $\{ \text{‘boundary’}, \{ \text{‘none’} \} \}$).

•{ ‘explode’, coef }

Build an ‘exploded’ view of the mesh: each convex is shrinked ($0 < \text{coef} \leq 1$). In the case of 3D convexes, only their faces are kept.

- { ‘union’, SLICEOP1, SLICEOP2 }
- { ‘intersection’, SLICEOP1, SLICEOP2 }
- { ‘comp’, SLICEOP }
- { ‘diff’, SLICEOP1, SLICEOP2 }

Boolean operations: returns the union,intersection,complementary or difference of slicing operations.

•{ ‘mesh’, MESH }

Build a slice which is the intersection of the sliced mesh with another mesh. The slice is such that all of its simplexes are stricly contained into a convex of each mesh.

EXAMPLE:

```
sl = gf_slice({'intersection',{'planar',+1,[0;0;0],[0;0;1]},... {'isovalues',-1,mf2,U2,0}},mf,U,5);
```

view the convex quality of a 2D or 3D mesh m:

```
gf_plot_slice(gfSlice({'explode', 0.7}, m, 2), 'convex_data',... gf_mesh_get(m,'quality'));
```

SPECIAL SLICES:

There are also some special calls to `gf_slice`:

- `gf_slice('streamlines',mf, U, mat SEEDS)`

compute streamlines of the (vector) field `U`, with seed points given by the columns of `SEEDS`.

- `gf_slice('points', m, mat PTS)`

return the “slice” composed of points given by the columns of `PTS` (useful for interpolation on a given set of sparse points, see `gf_compute(mf,U,'interpolate on',sl)`).

- `gf_slice('load', filename [,m])`

load the slice (and its `linked_mesh` if it is not given as an argument) from a text file.

General constructor for Slice objects

- `sl = Slice(sliceop, {Slice sl|{Mesh m| MeshFem mf, vec U}, int refine)[, mat CVfids])` Create a `@sl` using *sliceop* operation.

sliceop operation is specified with Tuple or List, do not forget the extra parentheses!. The first element is the name of the operation, followed the slicing options:

- (`'none'`) Does not cut the mesh.

- (`'planar', int orient, vec p, vec n`) Planar cut. p and n define a half-space, p being a point belong to the boundary of the half-space, and n being its normal. If *orient* is equal to -1 (resp. 0, +1), then the slicing operation will cut the mesh with the “interior” (resp. “boundary”, “exterior”) of the half-space. *orient* may also be set to +2 which means that the mesh will be sliced, but both the outer and inner parts will be kept.

- (`'ball', int orient, vec c, scalar r`) Cut with a ball of center c and radius r .

- (`'cylinder', int orient, vec p1, vec p2, scalar r`) Cut with a cylinder whose axis is the line $(p1,p2)$ and whose radius is r .

- (`'isovalues', int orient, MeshFem mf, vec U, scalar V`) Cut using the isosurface of the field U (defined on the `MeshFem mf`). The result is the set $\{x \text{ such that } U(x) \leq V\}$ or $\{x \text{ such that } U(x)=V\}$ or $\{x \text{ such that } U(x) \geq V\}$ depending on the value of *orient*.

- (`'boundary',[ SLICEOP]`) Return the boundary of the result of `SLICEOP`, where `SLICEOP` is any slicing operation. If `SLICEOP` is not specified, then the whole mesh is considered (i.e. it is equivalent to `('boundary',{ 'none' })`).

- (`'explode', mat Coef`) Build an ‘exploded’ view of the mesh: each convex is shrunk ($0 < \text{Coef} \leq 1$). In the case of 3D convexes, only their faces are kept.

- (`'union', SLICEOP1, SLICEOP2`)

- (`'intersection', SLICEOP1, SLICEOP2`)

- (`'diff', SLICEOP1, SLICEOP2`)

- (`'comp', SLICEOP`) Boolean operations: returns the union,intersection, difference or complementary of slicing operations.

- (`'mesh', Mesh m`) Build a slice which is the intersection of the sliced mesh with another mesh. The slice is such that all of its simplexes are stricly contained into a convex of each mesh.

- `sl = Slice('streamlines', MeshFem mf, mat U, mat Seeds)` Compute streamlines of the (vector) field U , with seed points given by the columns of *Seeds*.

- `sl = Slice('points', Mesh m, mat Pts)` Return the “slice” composed of points given by the columns of *Pts* (useful for interpolation on a given set of sparse points, see `gf_compute('interpolate on',sl)`).
- `sl = Slice('load', string filename[, Mesh m])` Load the slice (and its linked mesh if it is not given as an argument) from a text file.

area()

Return the area of the slice.

char()

Output a (unique) string representation of the Slice.

This can be used to perform comparisons between two different Slice objects. This function is to be completed.

cvs()

Return the list of convexes of the original mesh contained in the slice.

dim()

Return the dimension of the slice (2 for a 2D mesh, etc..).

display()

displays a short summary for a Slice object.

edges()

Return the edges of the linked mesh contained in the slice.

P contains the list of all edge vertices, *E1* contains the indices of each mesh edge in *P*, and *E2* contains the indices of each “edges” which is on the border of the slice. This function is useless except for post-processing purposes.

export_to_dx(filename, *args)

Synopsis: `Slice.export_to_dx(self, string filename, ...)`

Export a slice to OpenDX.

Following the *filename*, you may use any of the following options:

- if ‘ascii’ is not used, the file will contain binary data (non portable, but fast).
- if ‘edges’ is used, the edges of the original mesh will be written instead of the slice content.
- if ‘append’ is used, the opendx file will not be overwritten, and the new data will be added at the end of the file.

More than one dataset may be written, just list them. Each dataset consists of either:

- a field interpolated on the slice (scalar, vector or tensor), followed by an optional name.
- a `mesh_fem` and a field, followed by an optional name.

export_to_pos(filename, name=None, *args)

Synopsis: `Slice.export_to_pos(self, string filename[, string name][[,MeshFem mf1], mat U1, string nameU1[,MeshFem mf1], mat U2, string nameU2,...])`

Export a slice to Gmsh.

More than one dataset may be written, just list them. Each dataset consists of either:

- a field interpolated on the slice (scalar, vector or tensor).
- a `mesh_fem` and a field.

export_to_pov(filename)

Export a the triangles of the slice to POV-RAY.

export_to_vtk(filename, *args)

Synopsis: `Slice.export_to_vtk(self, string filename, ...)`

Export a slice to VTK.

Following the *filename*, you may use any of the following options:

- if ‘ascii’ is not used, the file will contain binary data (non portable, but fast).

- if 'edges' is used, the edges of the original mesh will be written instead of the slice content.

More than one dataset may be written, just list them. Each dataset consists of either:

- a field interpolated on the slice (scalar, vector or tensor), followed by an optional name.
- a mesh_fem and a field, followed by an optional name.

Examples:

- Slice.export_to_vtk('test.vtk', Usl, 'first_dataset', mf, U2, 'second_dataset')
- Slice.export_to_vtk('test.vtk', 'ascii', mf,U2)
- Slice.export_to_vtk('test.vtk', 'edges', 'ascii', Uslice)

interpolate_convex_data (*Ucv*)

Interpolate data given on each convex of the mesh to the slice nodes.

The input array *Ucv* may have any number of dimensions, but its last dimension should be equal to Mesh.max_cvid().

Example of use: Slice.interpolate_convex_data(Mesh.quality()).

linked_mesh ()

Return the mesh on which the slice was taken.

memsize ()

Return the amount of memory (in bytes) used by the slice object.

mesh ()

Return the mesh on which the slice was taken (identical to 'linked mesh')

nbpts ()

Return the number of points in the slice.

nbsplxs (*dim=None*)

Return the number of simplexes in the slice.

Since the slice may contain points (simplexes of dim 0), segments (simplexes of dimension 1), triangles etc., the result is a vector of size Slice.dim()+1, except if the optional argument *dim* is used.

pts ()

Return the list of point coordinates.

set_pts (*P*)

Replace the points of the slice.

The new points *P* are stored in the columns the matrix. Note that you can use the function to apply a deformation to a slice, or to change the dimension of the slice (the number of rows of *P* is not required to be equal to Slice.dim()).

splxs (*dim*)

Return the list of simplexes of dimension *dim*.

On output, S has 'dim+1' rows, each column contains the point numbers of a simplex. The vector CV2S can be used to find the list of simplexes for any convex stored in the slice. For example 'S[:,CV2S[4]:CV2S[5]]' gives the list of simplexes for the fourth convex.

7.17 Spmat

class Spmat (*args)

GetFEM Spmat object

Create a new sparse matrix in Getfem format. These sparse matrix can be stored as CSC (compressed column sparse), which is the format used by Matlab, or they can be stored as WSC (internal format to getfem). The CSC matrices are not writable (it would be very inefficient), but they are optimized for multiplication with vectors, and memory usage. The WSC are writable, they are very fast with respect to random read/write operation.

However their memory overhead is higher than CSC matrices, and they are a little bit slower for matrix-vector multiplications.

By default, all newly created matrices are build as WSC matrices. This can be changed later with `Spmat.to_csc(...)`, or may be changed automatically by `getfem` (for example `gf_linsolve()` converts the matrices to CSC).

The matrices may store REAL or COMPLEX values.

General constructor for Spmat objects

- `Spmat('empty', int m [, int n])` Create a new empty (i.e. full of zeros) sparse matrix, of dimensions $m \times n$. If n is omitted, the matrix dimension is $m \times m$.
- `Spmat('copy', mat K [, list I [, list J]])` Duplicate a matrix K (which might be a SpMat). If (index) I and/or J are given, the matrix will be a submatrix of K . For example:
`M = Spmat('copy', Spmat('empty',50,50), range(40), [6, 7, 8, 3, 10])` will return a 40x5 matrix.
- `Spmat('identity', int n)` Create a $n \times n$ identity matrix.
- `Spmat('mult', Spmat A, Spmat B)` Create a sparse matrix as the product of the sparse matrices A and B . It requires that A and B be both real or both complex, you may have to use `Spmat.to_complex()`
- `Spmat('add', Spmat A, Spmat B)` Create a sparse matrix as the sum of the sparse matrices A and B . Adding a real matrix with a complex matrix is possible.
- `Spmat('diag', mat D [, ivec E [, int n [,int m]])` Create a diagonal matrix. If E is given, D might be a matrix and each column of E will contain the sub-diagonal number that will be filled with the corresponding column of D .
- `Spmat('load', 'hb' | 'harwell-boeing' | 'mm' | 'matrix-market', string filename)` Read a sparse matrix from an Harwell-Boeing or a Matrix-Market file.

add (I, J, V)

Add V to the sub-matrix ' $M(I,J)$ '.

V might be a sparse matrix or a full matrix.

assign (I, J, V)

Copy V into the sub-matrix ' $M(I,J)$ '.

V might be a sparse matrix or a full matrix.

char ()

Output a (unique) string representation of the Spmat.

This can be used to perform comparisons between two different Spmat objects. This function is to be completed.

clear ($I=None, *args$)

Synopsis: `Spmat.clear(self[, list I[, list J]])`

Erase the non-zero entries of the matrix.

The optional arguments I and J may be specified to clear a sub-matrix instead of the entire matrix.

conjugate ()

Conjugate each element of the matrix.

csc_ind ()

Return the two usual index arrays of CSC storage.

If M is not stored as a CSC matrix, it is converted into CSC.

csc_val ()

Return the array of values of all non-zero entries of M .

If M is not stored as a CSC matrix, it is converted into CSC.

diag ($E=None$)

Return the diagonal of M as a vector.

If E is used, return the sub-diagonals whose ranks are given in E .

dirichlet_nullspace (*R*)

Solve the dirichlet conditions $M.U=R$.

A solution $U0$ which has a minimum L2-norm is returned, with a sparse matrix N containing an orthogonal basis of the kernel of the (assembled) constraints matrix M (hence, the PDE linear system should be solved on this subspace): the initial problem

$K.U = B$ with constraints $M.U = R$

is replaced by

$(N'.K.N).UU = N'.B$ with $U = N.UU + U0$

display ()

displays a short summary for a Spmat object.

full (*I=None, *args*)

Synopsis: $Sm = \text{Spmat.full}(\text{self}[, \text{list } I[, \text{list } J]])$

Return a full (sub-)matrix.

The optional arguments *I* and *J*, are the sub-intervals for the rows and columns that are to be extracted.

is_complex ()

Return 1 if the matrix contains complex values.

mult (*V*)

Product of the sparse matrix M with a vector V .

For matrix-matrix multiplications, see Spmat('mult').

nnz ()

Return the number of non-null values stored in the sparse matrix.

save (*format, filename*)

Export the sparse matrix.

the format of the file may be 'hb' for Harwell-Boeing, or 'mm' for Matrix-Market.

scale (*v*)

Multiplies the matrix by a scalar value *v*.

set_diag (*D, E=None*)

Change the diagonal (or sub-diagonals) of the matrix.

If *E* is given, *D* might be a matrix and each column of *E* will contain the sub-diagonal number that will be filled with the corresponding column of *D*.

size ()

Return a vector where *ni* and *nj* are the dimensions of the matrix.

storage ()

Return the storage type currently used for the matrix.

The storage is returned as a string, either 'CSC' or 'WSC'.

tmult (*V*)

Product of M transposed (conjugated if M is complex) with the vector V .

to_complex ()

Store complex numbers.

to_csc ()

Convert the matrix to CSC storage.

CSC storage is recommended for matrix-vector multiplications.

to_wsc ()

Convert the matrix to WSC storage.

Read and write operation are quite fast with WSC storage.

transconj ()

Transpose and conjugate the matrix.

transpose ()

Transpose the matrix.

7.18 Module asm

General assembly function.

Many of the functions below use more than one mesh_fem: the main mesh_fem (mf_u) used for the main unknow, and data mesh_fem (mf_d) used for the data. It is always assumed that the Qdim of mf_d is equal to 1: if mf_d is used to describe vector or tensor data, you just have to “stack” (in fortran ordering) as many scalar fields as necessary.

asm_mass_matrix (*mim, mf1, mf2=None*)

Assembly of a mass matrix.

Return a SpMat object.

asm_laplacian (*mim, mf_u, mf_d, a*)

Assembly of the matrix for the Laplacian problem.

$\nabla \cdot (a(x) \nabla u)$ with a a scalar.

Return a SpMat object.

asm_linear_elasticity (*mim, mf_u, mf_d, lambda_d, mu_d*)

Assembles of the matrix for the linear (isotropic) elasticity problem.

$\nabla \cdot (C(x) : \nabla u)$ with C defined via $lambda_d$ and mu_d .

Return a SpMat object.

asm_nonlinear_elasticity (*mim, mf_u, U, law, mf_d, params, *args*)

Synopsis: TRHS = asm_nonlinear_elasticity(MeshIm mim, MeshFem mf_u, vec U, string law, MeshFem mf_d, mat params, {'tangent matrix' | 'rhs' | 'incompressible tangent matrix', MeshFem mf_p, vec Pl' | 'incompressible rhs', MeshFem mf_p, vec P})

Assembles terms (tangent matrix and right hand side) for nonlinear elasticity.

The solution U is required at the current time-step. The *law* may be choosen among:

- ‘Saint Venant Kirchhoff’: Linearized law, should be avoided). This law has the two usual Lamé coefficients as parameters, called λ and μ .
- ‘Mooney Rivlin’: Only for incompressibility. This law has two parameters, called $C1$ and $C2$.
- ‘Ciarlet Geymonat’: This law has 3 parameters, called λ , μ and γ , with γ chosen such that γ is in $]-\lambda/2 - \mu, -\mu[$.

The parameters of the material law are described on the MeshFem *mf_d*. The matrix *params* should have *nbdof(mf_d)* columns, each row corresponds to a parameter.

The last argument selects what is to be built: either the tangent matrix, or the right hand side. If the incompressibility is considered, it should be followed by a MeshFem *mf_p*, for the pression.

Return a SpMat object (tangent matrix), vec object (right hand side), tuple of SpMat objects (incompressible tangent matrix), or tuple of vec objects (incompressible right hand side).

asm_stokes (*mim, mf_u, mf_p, mf_d, nu*)

Assembly of matrices for the Stokes problem.

$-\nu(x) \Delta u + \nabla p = 0$ $\nabla \cdot u = 0$ with ν (*nu*), the fluid’s dynamic viscosity.

On output, K is the usual linear elasticity stiffness matrix with $\lambda = 0$ and $2\mu = \nu$. B is a matrix corresponding to $\int p \nabla \cdot \phi$.

K and B are SpMat object’s.

asm_helmholtz (*mim, mf_u, mf_d, k*)

Assembly of the matrix for the Helmholtz problem.

$\Delta u + k^2 u = 0$, with k complex scalar.

Return a SpMat object.

asm_bilaplacian (*mim, mf_u, mf_d, a*)

Assembly of the matrix for the Bilaplacian problem.

$\Delta(a(x)\Delta u) = 0$ with a scalar.

Return a SpMat object.

asm_volumic_source (*mim, mf_u, mf_d, fd*)

Assembly of a volumic source term.

Output a vector V , assembled on the MeshFem mf_u , using the data vector fd defined on the data MeshFem mf_d . fd may be real or complex-valued.

Return a vec object.

asm_boundary_source (*bnum, mim, mf_u, mf_d, G*)

Assembly of a boundary source term.

G should be a $[Qdim \times N]$ matrix, where N is the number of dof of mf_d , and $Qdim$ is the dimension of the unknown u (that is set when creating the MeshFem).

Return a vec object.

asm_dirichlet (*bnum, mim, mf_u, mf_d, H, R, threshold=None*)

Assembly of Dirichlet conditions of type $h.u = r$.

Handle $h.u = r$ where h is a square matrix (of any rank) whose size is equal to the dimension of the unknown u . This matrix is stored in H , one column per dof in mf_d , each column containing the values of the matrix h stored in fortran order:

$$H(:, j) = [h_{11}(x_j) h_{21}(x_j) h_{12}(x_j) h_{22}(x_j)]^t$$

if u is a 2D vector field.

Of course, if the unknown is a scalar field, you just have to set $H = \text{ones}(1, N)$, where N is the number of dof of mf_d .

This is basically the same than calling `gf_asm('boundary qu term')` for H and calling `gf_asm('neumann')` for R , except that this function tries to produce a 'better' (more diagonal) constraints matrix (when possible).

See also `SpMat.Dirichlet_nullspace()`.

asm_boundary_qu_term (*boundary_num, mim, mf_u, mf_d, q*)

Assembly of a boundary qu term.

q should be a $[Qdim \times Qdim \times N]$ array, where N is the number of dof of mf_d , and $Qdim$ is the dimension of the unknown u (that is set when creating the MeshFem).

Return a SpMat object.

asm_volumic (*CVLST=None, *args*)

Synopsis: (...) = `asm_volumic(CVLST, expr [, mesh_ims, mesh_fems, data...])`

Generic assembly procedure for volumic assembly.

The expression *expr* is evaluated over the MeshFem's listed in the arguments (with optional data) and assigned to the output arguments. For details about the syntax of assembly expressions, please refer to the getfem user manual (or look at the file `getfem_assembling.h` in the getfem++ sources).

For example, the L2 norm of a field can be computed with:

`gf_compute('L2 norm')` or with:

```
gf_asm('volumic', 'u=data(#1); V()+=u(i).u(j).comp(Base(#1).Base(#1))(i,j)', mim, mf, U)
```

The Laplacian stiffness matrix can be evaluated with:

`gf_asm('laplacian', mim, mf, A)` or equivalently with:

```
gf_asm('volumic', 'a=data(#2); M(#1, #1) += sym(comp(Grad(#1).Grad(#1).Base(#2))(:, i, :, i, j).a(j))', m
```

asm_boundary(*bnum, expr, mim=None, mf=None, data=None, *args*)

Synopsis: (...) = `asm_boundary`(int *bnum*, string *expr* [, `MeshIm` *mim*, `MeshFem` *mf*, *data*...])

Generic boundary assembly.

See the help for `gf_asm('volumic')`.

asm_interpolation_matrix(*mf, mfi*)

Build the interpolation matrix from a `MeshFem` onto another `MeshFem`.

Return a matrix M_i , such that $V = M_i.U$ is equal to `gf_compute('interpolate_on', mfi)`. Useful for repeated interpolations. Note that this is just interpolation, no elementary integrations are involved here, and *mfi* has to be lagrangian. In the more general case, you would have to do a L2 projection via the mass matrix.

M_i is a `SpMat` object.

asm_extrapolation_matrix(*mf, mfe*)

Build the extrapolation matrix from a `MeshFem` onto another `MeshFem`.

Return a matrix M_e , such that $V = M_e.U$ is equal to `gf_compute('extrapolate_on', mfe)`. Useful for repeated extrapolations.

M_e is a `SpMat` object.

7.19 Module compute

Various computations involving the solution U to a finite element problem.

compute_L2_norm(*MF, U, mim, CVids=None*)

Compute the L2 norm of the (real or complex) field U .

If *CVids* is given, the norm will be computed only on the listed convexes.

compute_H1_semi_norm(*MF, U, mim, CVids=None*)

Compute the L2 norm of $\text{grad}(U)$.

If *CVids* is given, the norm will be computed only on the listed convexes.

compute_H1_norm(*MF, U, mim, CVids=None*)

Compute the H1 norm of U .

If *CVids* is given, the norm will be computed only on the listed convexes.

compute_H2_semi_norm(*MF, U, mim, CVids=None*)

Compute the L2 norm of $D^2(U)$.

If *CVids* is given, the norm will be computed only on the listed convexes.

compute_H2_norm(*MF, U, mim, CVids=None*)

Compute the H2 norm of U .

If *CVids* is given, the norm will be computed only on the listed convexes.

compute_gradient (*MF, U, mf_du*)

Compute the gradient of the field *U* defined on MeshFem *mf_du*.

The gradient is interpolated on the MeshFem *mf_du*, and returned in *DU*. For example, if *U* is defined on a P2 MeshFem, *DU* should be evaluated on a P1-discontinuous MeshFem. *mf* and *mf_du* should share the same mesh.

U may have any number of dimensions (i.e. this function is not restricted to the gradient of scalar fields, but may also be used for tensor fields). However the last dimension of *U* has to be equal to the number of dof of *mf*. For example, if *U* is a [3x3xNmf] array (where Nmf is the number of dof of *mf*), *DU* will be a [Nx3x3[xQ]xNmf_du] array, where N is the dimension of the mesh, Nmf_du is the number of dof of *mf_du*, and the optional Q dimension is inserted if *Qdim_mf* != *Qdim_mf_du*, where *Qdim_mf* is the Qdim of *mf* and *Qdim_mf_du* is the Qdim of *mf_du*.

compute_hessian (*MF, U, mf_h*)

Compute the hessian of the field *U* defined on MeshFem *mf_h*.

See also `gf_compute('gradient', MeshFem mf_du)`.

compute_eval_on_triangulated_surface (*MF, U, Nrefine, CVLIST=None*)

[OBSOLETE FUNCTION! will be removed in a future release] Utility function designed for 2D triangular meshes : returns a list of triangles coordinates with interpolated U values. This can be used for the accurate visualization of data defined on a discontinuous high order element. On output, the six first rows of UP contains the triangle coordinates, and the others rows contain the interpolated values of U (one for each triangle vertex) CVLIST may indicate the list of convex number that should be consider, if not used then all the mesh convexes will be used. U should be a row vector.

compute_interpolate_on (*MF, U, *args*)

Synopsis: `Ui = compute_interpolate_on(MeshFem MF, vec U, {MeshFem mfi | Slice sli})`

Interpolate a field on another MeshFem or a Slice.

- Interpolation on another MeshFem *mfi***: *mfi* has to be Lagrangian. If *mf* and *mfi* share the same mesh object, the interpolation will be much faster.

- Interpolation on a Slice *sli***: this is similar to interpolation on a refined P1-discontinuous mesh, but it is much faster. This can also be used with `Slice('points')` to obtain field values at a given set of points.

See also `gf_asm('interpolation matrix')`

compute_extrapolate_on (*MF, U, mfe*)

Extrapolate a field on another MeshFem.

If the mesh of *mfe* is stricly included in the mesh of *mf*, this function does stricly the same job as `gf_compute('interpolate_on')`. However, if the mesh of *mfe* is not exactly included in *mf* (imagine interpolation between a curved refined mesh and a coarse mesh), then values which are outside *mf* will be extrapolated.

See also `gf_asm('extrapolation matrix')`

compute_error_estimate (*MF, U, mim*)

Compute an a posteriori error estimate.

Currently there is only one which is available: for each convex, the jump of the normal derivative is integrated on its faces.

compute_convect (*MF, U, mf_v, V, dt, nt, option=None*)

Compute a convection of *U* with regards to a steady state velocity field *V* with a Characteristic-Galerkin method. This method is restricted to pure Lagrange fems for *U*. *mf_v* should represent a continuous finite element method. *dt* is the integration time and *nt* is the number of integration step on the characteristics. *option* is an option for the part of the boundary where there is a re-entrant convection. *option* = 'extrapolation' for an extrapolation on the nearest element or *option* = 'unchanged' for a constant value on that boundary. This method is rather dissipative, but stable.

7.20 Module delete

Delete an existing getfem object from memory (mesh, mesh_fem, etc.).

SEE ALSO: gf_workspace, gf_mesh, gf_mesh_fem.

delete (*I*, *J=None*, *K=None*, *args)

Synopsis: delete([I, J, K,...])

I should be a descriptor given by gf_mesh(), gf_mesh_im(), gf_slice() etc.

Note that if another object uses *I*, then object *I* will be deleted only when both have been asked for deletion.

Only objects listed in the output of gf_workspace('stats') can be deleted (for example gf_fem objects cannot be destroyed).

You may also use gf_workspace('clear all') to erase everything at once.

7.21 Module linsolve

Various linear system solvers.

linsolve_gmres (*M*, *b*, *restart=None*, *args)

Synopsis: X = linsolve_gmres(SpMat *M*, vec *b* [, int *restart*] [, Mrecond *P*] [, 'noisy'] [, 'res', *r*] [, 'maxiter', *n*])

Solve $MX = b$ with the generalized minimum residuals method.

Optionally using *P* as preconditioner. The default value of the restart parameter is 50.

linsolve_cg (*M*, *b*, *P=None*, *args)

Synopsis: X = linsolve_cg(SpMat *M*, vec *b* [, Mrecond *P*] [, 'noisy'] [, 'res', *r*] [, 'maxiter', *n*])

Solve $MX = b$ with the conjugated gradient method.

Optionally using *P* as preconditioner.

linsolve_bicgstab (*M*, *b*, *P=None*, *args)

Synopsis: X = linsolve_bicgstab(SpMat *M*, vec *b* [, Mrecond *P*] [, 'noisy'] [, 'res', *r*] [, 'maxiter', *n*])

Solve $MX = b$ with the bi-conjugated gradient stabilized method.

Optionally using *P* as a preconditioner.

linsolve_lu (*M*, *b*)

Alias for gf_linsolve('superlu',...)

linsolve_superlu (*M*, *b*)

Solve $M.U = b$ apply the SuperLU solver (sparse LU factorization).

The condition number estimate *cond* is returned with the solution *U*.

7.22 Module poly

Performs various operations on the polynom POLY.

poly_print (*P*)

Prints the content of *P*.

poly_product (*P*)

To be done ... !

7.23 Module undelete

Undelete an existing getfem object from memory (mesh, mesh_fem, etc.).

SE ALSO: gf_workspace, gf_delete.

undelete (*I*, *J=None*, *K=None*, **args*)

Synopsis: undelete(I[, J, K,...])

I should be a descriptor given by gf_mesh(), gf_mesh_im(), gf_slice() etc.

7.24 Module util

Performs various operations which do not fit elsewhere.

util_save_matrix (*FMT*, *FILENAME*, *A*)

Exports a sparse matrix into the file named FILENAME, using Harwell-Boeing (FMT='hb') or Matrix-Market (FMT='mm') formatting.

util_load_matrix (*FMT*, *FILENAME*)

Imports a sparse matrix from a file.

util_trace_level (*level*)

Set the verbosity of some getfem++ routines.

Typically the messages printed by the model bricks, 0 means no trace message (default is 3).

util_warning_level (*level*)

Filter the less important warnings displayed by getfem.

0 means no warnings, default level is 3.

INDEX

A

`adapt()` (MeshIm method), 49
`adapt()` (MeshLevelSet method), 50
`add()` (MeshLevelSet method), 50
`add()` (Spmat method), 63
`add_basic_contact_brick()` (Model method), 51
`add_basic_d2_on_dt2_brick()` (Model method), 52
`add_basic_d_on_dt_brick()` (Model method), 52
`add_constraint_with_multipliers()` (Model method), 52
`add_constraint_with_penalization()` (Model method), 52
`add_contact_with_rigid_obstacle_brick()` (Model method), 52
`add_convex()` (Mesh method), 40
`add_data()` (Model method), 53
`add_Dirichlet_condition_with_multipliers()` (Model method), 51
`add_Dirichlet_condition_with_penalization()` (Model method), 51
`add_explicit_matrix()` (Model method), 53
`add_explicit_rhs()` (Model method), 53
`add_fem_data()` (Model method), 53
`add_fem_variable()` (Model method), 53
`add_Fourier_Robin_brick()` (Model method), 51
`add_generalized_Dirichlet_condition_with_multipliers()` (Model method), 53
`add_generalized_Dirichlet_condition_with_penalization()` (Model method), 53
`add_generic_elliptic_brick()` (Model method), 53
`add_Helmholtz_brick()` (Model method), 51
`add_initialized_data()` (Model method), 54
`add_initialized_fem_data()` (Model method), 54
`add_isotropic_linearized_elasticity_brick()` (Model method), 54
`add_Laplacian_brick()` (Model method), 51
`add_linear_incompressibility_brick()` (Model method), 54
`add_mass_brick()` (Model method), 54
`add_midpoint_dispatcher()` (Model method), 54
`add_multiplier()` (Model method), 54
`add_nonlinear_elasticity_brick()` (Model method), 54

`add_nonlinear_incompressibility_brick()` (Model method), 54
`add_normal_source_term_brick()` (Model method), 55
`add_point()` (Mesh method), 40
`add_source_term_brick()` (Model method), 55
`add_theta_method_dispatcher()` (Model method), 55
`add_unilateral_contact_brick()` (Model method), 55
`add_variable()` (Model method), 55
`area()` (Slice method), 61
`asm_bilaplacian()` (in module `getfem`), 66
`asm_boundary()` (in module `getfem`), 67
`asm_boundary_qu_term()` (in module `getfem`), 66
`asm_boundary_source()` (in module `getfem`), 66
`asm_dirichlet()` (in module `getfem`), 66
`asm_extrapolation_matrix()` (in module `getfem`), 67
`asm_helmholtz()` (in module `getfem`), 65
`asm_interpolation_matrix()` (in module `getfem`), 67
`asm_laplacian()` (in module `getfem`), 65
`asm_linear_elasticity()` (in module `getfem`), 65
`asm_mass_matrix()` (in module `getfem`), 65
`asm_nonlinear_elasticity()` (in module `getfem`), 65
`asm_stokes()` (in module `getfem`), 65
`asm_volumic()` (in module `getfem`), 66
`asm_volumic_source()` (in module `getfem`), 66
`assembly`, 7
`assembly()` (Model method), 55
`assign()` (Spmat method), 63

B

`base_value()` (Fem method), 29
`basic_dof_from_cv()` (MeshFem method), 44
`basic_dof_from_cvid()` (MeshFem method), 45
`basic_dof_nodes()` (MeshFem method), 45
`basic_dof_on_region()` (MeshFem method), 45
`basic_structure()` (CvStruct method), 27
`boundaries()` (Mesh method), 40
`boundary()` (Mesh method), 40

C

`change_penalization_coeff()` (Model method), 55
`char()` (CvStruct method), 27

char() (Fem method), 29
 char() (GeoTrans method), 30
 char() (GlobalFunction method), 31
 char() (Integ method), 32
 char() (LevelSet method), 33
 char() (MdBrick method), 36
 char() (MdState method), 38
 char() (Mesh method), 40
 char() (MeshFem method), 45
 char() (MeshIm method), 49
 char() (MeshLevelSet method), 50
 char() (Model method), 56
 char() (Precond method), 58
 char() (Slice method), 61
 char() (Spmat method), 63
 clear() (MdState method), 38
 clear() (Model method), 56
 clear() (Spmat method), 63
 coeffs() (Integ method), 32
 compute_convect() (in module getfem), 68
 compute_error_estimate() (in module getfem), 68
 compute_eval_on_triangulated_surface() (in module getfem), 68
 compute_extrapolate_on() (in module getfem), 68
 compute_gradient() (in module getfem), 67
 compute_H1_norm() (in module getfem), 67
 compute_H1_semi_norm() (in module getfem), 67
 compute_H2_norm() (in module getfem), 67
 compute_H2_semi_norm() (in module getfem), 67
 compute_hessian() (in module getfem), 68
 compute_interpolate_on() (in module getfem), 68
 compute_isotropic_linearized_Von_Mises_or_Tresca() (Model method), 56
 compute_L2_norm() (in module getfem), 67
 compute_reduced_residual() (MdState method), 38
 compute_reduced_system() (MdState method), 38
 compute_residual() (MdState method), 38
 compute_tangent_matrix() (MdState method), 38
 compute_Von_Mises_or_Tresca() (Model method), 56
 conjugate() (Spmat method), 63
 constraints() (MdBrick method), 36
 constraints_matrix() (MdState method), 38
 constraints_nullspace() (MdState method), 38
 constraints_rhs() (MdBrick method), 36
 contact_brick_set_BN() (Model method), 56
 contact_brick_set_BT() (Model method), 56
 convex id, 8
 convex_area() (Mesh method), 40
 convex_index() (MeshFem method), 45
 convex_index() (MeshIm method), 49
 convexes, 7
 crack_tip_convexes() (MeshLevelSet method), 50
 csc_ind() (Spmat method), 63
 csc_val() (Spmat method), 63

curved_edges() (Mesh method), 40
 cut_mesh() (MeshLevelSet method), 50
 cvid, 8
 cvid() (Mesh method), 40
 cvid_from_pid() (Mesh method), 40
 cvs() (Slice method), 61
 CvStruct (built-in class), 10
 CvStruct (class in getfem), 27
 cvstruct() (Mesh method), 40

D

degree() (LevelSet method), 33
 del_convex() (Mesh method), 40
 del_convex_of_dim() (Mesh method), 40
 del_point() (Mesh method), 41
 delete() (in module getfem), 69
 delete_boundary() (Mesh method), 41
 delete_region() (Mesh method), 41
 diag() (Spmat method), 63
 dim() (CvStruct method), 27
 dim() (Fem method), 29
 dim() (GeoTrans method), 30
 dim() (Integ method), 32
 dim() (MdBrick method), 36
 dim() (Mesh method), 41
 dim() (Slice method), 61
 dirichlet_nullspace() (Spmat method), 64
 disable_bricks() (Model method), 56
 display() (CvStruct method), 27
 display() (Fem method), 29
 display() (GeoTrans method), 30
 display() (GlobalFunction method), 31
 display() (Integ method), 32
 display() (LevelSet method), 33
 display() (MdBrick method), 36
 display() (MdState method), 38
 display() (Mesh method), 41
 display() (MeshFem method), 45
 display() (MeshIm method), 49
 display() (MeshLevelSet method), 50
 display() (Model method), 56
 display() (Precond method), 58
 display() (Slice method), 61
 display() (Spmat method), 64
 dof, 7
 dof_from_cv() (MeshFem method), 45
 dof_from_cvid() (MeshFem method), 45
 dof_from_im() (MeshFem method), 45
 dof_nodes() (MeshFem method), 45
 dof_on_region() (MeshFem method), 45
 dof_partition() (MeshFem method), 46

E

edges() (Mesh method), 41

edges() (Slice method), 61
 Eltm (class in getfem), 27
 eltm() (MeshIm method), 49
 environment variable
 assembly, 7
 convex id, 8
 convexes, 7
 cvid, 8
 dof, 7
 FEM, 7
 geometric transformation, 7
 geometrical nodes, 7
 integration method, 7
 interpolate, 7
 Lagrangian, 7
 mesh, 7
 mesh nodes, 7
 mesh_fem, 7
 mesh_im, 7
 pid, 8
 point id, 8
 Qdim, 7
 quadrature formulas, 7
 reference convex, 7
 estimated_degree() (Fem method), 29
 eval() (MeshFem method), 46
 export_to_dx() (Mesh method), 41
 export_to_dx() (MeshFem method), 46
 export_to_dx() (Slice method), 61
 export_to_pos() (Mesh method), 41
 export_to_pos() (MeshFem method), 46
 export_to_pos() (Slice method), 61
 export_to_pov() (Slice method), 61
 export_to_vtk() (Mesh method), 41
 export_to_vtk() (MeshFem method), 46
 export_to_vtk() (Slice method), 61
 extension_matrix() (MeshFem method), 46

F

face() (CvStruct method), 27
 face_coeffs() (Integ method), 32
 face_pts() (Integ method), 32
 facepts() (CvStruct method), 27
 faces_from_cvid() (Mesh method), 41
 faces_from_pid() (Mesh method), 41
 FEM, 7
 Fem (built-in class), 10
 Fem (class in getfem), 28
 fem() (MeshFem method), 46
 first_iter() (Model method), 56
 from_variables() (Model method), 56
 full() (Spmat method), 64

G

geometric transformation, 7
 geometrical nodes, 7
 GeoTrans (built-in class), 10
 GeoTrans (class in getfem), 30
 geotrans() (Mesh method), 41
 GlobalFunction (class in getfem), 31
 grad() (GlobalFunction method), 31
 grad_base_value() (Fem method), 29

H

has_linked_mesh_levelset() (MeshFem method), 46
 hess() (GlobalFunction method), 31
 hess_base_value() (Fem method), 29

I

im_nodes() (MeshIm method), 49
 Integ (built-in class), 10
 Integ (class in getfem), 31
 integ() (MeshIm method), 49
 integration method, 7
 interpolate, 7
 interpolate_convex_data() (MeshFem method), 46
 interpolate_convex_data() (Slice method), 62
 interval_of_variable() (Model method), 56
 is_coercive() (MdBrick method), 36
 is_complex() (MdBrick method), 36
 is_complex() (MdState method), 38
 is_complex() (Model method), 56
 is_complex() (Precond method), 58
 is_complex() (Spmat method), 64
 is_equivalent() (Fem method), 29
 is_equivalent() (MeshFem method), 47
 is_exact() (Integ method), 32
 is_lagrange() (Fem method), 29
 is_lagrangian() (MeshFem method), 47
 is_linear() (GeoTrans method), 30
 is_linear() (MdBrick method), 36
 is_polynomial() (Fem method), 29
 is_polynomial() (MeshFem method), 47
 is_reduced() (MeshFem method), 47
 is_symmetric() (MdBrick method), 36

L

Lagrangian, 7
 LevelSet (class in getfem), 33
 levelsets() (MeshLevelSet method), 50
 linked_mesh() (MeshFem method), 47
 linked_mesh() (MeshIm method), 49
 linked_mesh() (MeshLevelSet method), 50
 linked_mesh() (Slice method), 62
 linked_mesh_levelset() (MeshFem method), 47
 linsolve_bicgstab() (in module getfem), 69

linsolve_cg() (in module getfem), 69
 linsolve_gmres() (in module getfem), 69
 linsolve_lu() (in module getfem), 69
 linsolve_superlu() (in module getfem), 69
 listbricks() (Model method), 56
 listvar() (Model method), 56

M

matrix_term() (Model method), 56
 max_cvid() (Mesh method), 41
 max_pid() (Mesh method), 41
 MdBrick (built-in class), 10
 MdBrick (class in getfem), 34
 MdState (built-in class), 10
 MdState (class in getfem), 38
 memsize() (LevelSet method), 33
 memsize() (MdBrick method), 36
 memsize() (MdState method), 38
 memsize() (Mesh method), 42
 memsize() (MeshFem method), 47
 memsize() (MeshIm method), 49
 memsize() (MeshLevelSet method), 50
 memsize() (Model method), 56
 memsize() (Slice method), 62
 merge() (Mesh method), 42
 mesh, 7
 Mesh (built-in class), 10
 Mesh (class in getfem), 39
 mesh nodes, 7
 mesh() (MeshFem method), 47
 mesh() (Slice method), 62
 mesh_fem, 7
 mesh_im, 7
 MeshFem (built-in class), 10
 MeshFem (class in getfem), 44
 MeshIm (built-in class), 10
 MeshIm (class in getfem), 48
 MeshLevelSet (class in getfem), 50
 mf() (LevelSet method), 33
 mixed_variables() (MdBrick method), 36
 Model (built-in class), 10
 Model (class in getfem), 50
 mult() (Precond method), 58
 mult() (Spmat method), 64
 mult_varname_Dirichlet() (Model method), 56

N

nb_basic_dof() (MeshFem method), 47
 nb_constraints() (MdBrick method), 36
 nb_ls() (MeshLevelSet method), 50
 nbcv() (Mesh method), 42
 nbndof() (Fem method), 29
 nbndof() (MdBrick method), 36
 nbndof() (MeshFem method), 47

nbpts() (CvStruct method), 27
 nbpts() (GeoTrans method), 30
 nbpts() (Integ method), 32
 nbpts() (Mesh method), 42
 nbpts() (Slice method), 62
 nbsplxs() (Slice method), 62
 next_iter() (Model method), 56
 nnz() (Spmat method), 64
 non_conformal_basic_dof() (MeshFem method), 47
 non_conformal_dof() (MeshFem method), 47
 normal_of_face() (Mesh method), 42
 normal_of_faces() (Mesh method), 42
 normals() (GeoTrans method), 30

O

optimize_structure() (Mesh method), 42
 orphaned_pid() (Mesh method), 42
 outer_faces() (Mesh method), 42

P

param() (MdBrick method), 37
 param_list() (MdBrick method), 37
 penalization_epsilon() (MdBrick method), 37
 pid, 8
 pid() (Mesh method), 42
 pid_from_coords() (Mesh method), 42
 pid_from_cvid() (Mesh method), 42
 pid_in_cvids() (Mesh method), 42
 pid_in_faces() (Mesh method), 43
 pid_in_regions() (Mesh method), 43
 point id, 8
 poly_print() (in module getfem), 69
 poly_product() (in module getfem), 69
 poly_str() (Fem method), 29
 Precond (class in getfem), 57
 pts() (Fem method), 29
 pts() (GeoTrans method), 30
 pts() (Integ method), 33
 pts() (Mesh method), 43
 pts() (Slice method), 62
 pts_from_cvid() (Mesh method), 43

Q

Qdim, 7
 qdim() (MeshFem method), 47
 quadrature formulas, 7
 quality() (Mesh method), 43

R

reduced_residual() (MdState method), 38
 reduced_tangent_matrix() (MdState method), 38
 reduction() (MeshFem method), 47
 reduction_matrices() (MeshFem method), 47

reduction_matrix() (MeshFem method), 47
 reference convex, 7
 refine() (Mesh method), 43
 region() (Mesh method), 43
 region_intersect() (Mesh method), 43
 region_merge() (Mesh method), 43
 region_subtract() (Mesh method), 43
 regions() (Mesh method), 43
 residual() (MdState method), 38
 resize_variable() (Model method), 57
 rhs() (Model method), 57

S

save() (Mesh method), 43
 save() (MeshFem method), 47
 save() (MeshIm method), 49
 save() (Spmat method), 64
 scale() (Spmat method), 64
 set_boundary() (Mesh method), 43
 set_classical_discontinuous_fem() (MeshFem method), 48
 set_classical_fem() (MeshFem method), 48
 set_diag() (Spmat method), 64
 set_dof_partition() (MeshFem method), 48
 set_fem() (MeshFem method), 48
 set_integ() (MeshIm method), 49
 set_param() (MdBrick method), 37
 set_partial() (MeshFem method), 48
 set_private_matrix() (Model method), 57
 set_private_rhs() (Model method), 57
 set_pts() (Mesh method), 43
 set_pts() (Slice method), 62
 set_qdim() (MeshFem method), 48
 set_region() (Mesh method), 43
 set_state() (MdState method), 38
 set_values() (LevelSet method), 33
 set_variable() (Model method), 57
 simplify() (LevelSet method), 33
 size() (Precond method), 58
 size() (Spmat method), 64
 Slice (class in getfem), 58
 solve() (MdBrick method), 37
 solve() (Model method), 57
 splxs() (Slice method), 62
 Spmat (class in getfem), 62
 state() (MdState method), 38
 storage() (Spmat method), 64
 subclass() (MdBrick method), 37
 sup() (MeshLevelSet method), 50

T

tangent_matrix() (MdState method), 39
 tangent_matrix() (Model method), 57
 target_dim() (Fem method), 30

tmult() (Precond method), 58
 tmult() (Spmat method), 64
 to_complex() (Spmat method), 64
 to_csc() (Spmat method), 64
 to_variables() (Model method), 57
 to_wsc() (Spmat method), 64
 transconj() (Spmat method), 64
 transform() (GeoTrans method), 30
 transform() (Mesh method), 44
 translate() (Mesh method), 44
 transpose() (Spmat method), 64
 tresca() (MdBrick method), 37
 triangulated_surface() (Mesh method), 44
 type() (Precond method), 58

U

unable_bricks() (Model method), 57
 undelete() (in module getfem), 70
 unreduce() (MdState method), 39
 util_load_matrix() (in module getfem), 70
 util_save_matrix() (in module getfem), 70
 util_trace_level() (in module getfem), 70
 util_warning_level() (in module getfem), 70

V

val() (GlobalFunction method), 31
 values() (LevelSet method), 33
 variable() (Model method), 57
 velocity_update_for_Newmark_scheme() (Model method), 57
 velocity_update_for_order_two_theta_method() (Model method), 57
 von_mises() (MdBrick method), 37